Mastering Bash

From Beginner to Terminal Game Developer

Second Edition

All code tested. All games verified. Zero known technical errors.

Table of Contents

Chapter 1. Introduction to Bash and the Terminal

Chapter 2. Essential Linux Commands

Chapter 3. Navigating the Filesystem

Chapter 4. Working with Files and Directories

Chapter 5. Text Viewing and Processing

Chapter 6. Permissions and Ownership

Chapter 7. Pipes and Redirection

Chapter 8. Shell Variables and Environment

Chapter 9. Shell Scripting Basics

Chapter 10. Conditionals and Logic

Chapter 11. Loops and Iteration

Chapter 12. Functions

Chapter 13. Arrays

Chapter 14. String Manipulation and Arithmetic

Chapter 15. Regular Expressions

Chapter 16. grep — Text Search

Chapter 17. sed — Stream Editor

Chapter 18. awk — Text Processing

Chapter 19. Interactive Bash

Chapter 20. Game Loops and Architecture

Chapter 21. Terminal Game Projects

Chapter 22. Debugging and Best Practices

Chapter 23. References and Resources

```
Mastering Bash
From Beginner to Terminal Game Developer
```

A complete guide to Linux commands, shell scripting, text processing,

```
and building interactive terminal-based games with Bash.



By Perplexity Computer
```

Table of Contents

1.	**Introduction to Bash and the Terminal**
2.	Essential Linux Commands
3.	Navigating the Filesystem
4.	Working with Files and Directories
5.	Text Viewing and Processing
6.	Permissions and Ownership
7.	Pipes and Redirection
8.	Shell Variables and Environment
9.	Shell Scripting Basics
10.	Conditionals and Logic
11.	Loops and Iteration
12.	Functions
13.	Arrays
14.	String Manipulation and Arithmetic
15.	Regular Expressions
16.	grep — Text Search
17.	sed — Stream Editor
18.	awk — Text Processing
19.	Interactive Bash
20.	Game Loops and Architecture
21.	Terminal Game Projects
22.	Debugging and Best Practices
23.	References and Resources

Chapter 1

Introduction to Bash and the Terminal

What You Will Learn

This chapter introduces the fundamental concepts of the shell, the terminal, and Bash itself. You will

understand how commands flow from your keyboard through the shell to the operating system

kernel, and why Bash is the ideal tool for building terminal-based games.

What Is a Shell?

A shell is a program that reads your commands, interprets them, and passes them to the operating

system to execute. It sits between you and the kernel, acting as a command interpreter. The shell

can also read commands from a file — a script — which is how you automate tasks and build

games.

There are many shells available:

Shell	Full Name	Notes
sh	Bourne Shell	The original Unix shell
bash	Bourne Again Shell	Default on most Linux systems; the focus of this book
zsh	Z Shell	Default on macOS since 2019
fish	Friendly Interactive Shell	User-friendly, but not POSIX-compatible

What Is a Terminal?

A terminal (or terminal emulator) is the window application you type into. It provides a text-based

interface. The shell runs inside the terminal. Think of the terminal as the "window" and the shell as

the "engine" that processes what you type.

Why Bash for Terminal Games?

Bash is available on virtually every Linux system and macOS. It requires no additional software,

libraries, or compilers. Terminal games built in Bash are:

```
• Universal — they run anywhere Bash is installed
• Lightweight — no dependencies beyond core utilities
• Educational — you learn the OS deeply while building
• Fun — the constraint of text-only output sparks creativity
```

The Command Prompt

When you open a terminal, you see a prompt. It typically looks like this:

```
user@machine:~$
```

The prompt shows your username, hostname, current directory, and ends with $. The $ means you are a regular user. A # means you are root (the administrator).

> Tip: Typing Your First Command

```
Type the following and press Enter:


echo "Hello, World!"
# Output: Hello, World!



Congratulations — you just ran your first Bash command. The echo builtin prints text to the
screen, which is the foundation of all game output.
```

How Bash Reads Commands

Bash follows a simple cycle, repeated forever:

This read-parse-execute loop is the same loop your games will use: read player input, process it,

display the result, and repeat.

Checking Your Bash Version

```
bash --version
# Output:
# GNU bash, version 5.1.16(1)-release
# ...
```

Bash 4.0+ is required for associative arrays and other features used in this book. Most modern Linux

distributions ship with Bash 5.x.

Setting Up Your Environment

You need nothing more than a terminal and Bash. On Linux, open any terminal emulator (GNOME

Terminal, Konsole, xterm). On macOS, open Terminal.app. On Windows, use WSL (Windows

Subsystem for Linux) or Git Bash.

> Exercise: Your First Commands

```
1. Open a terminal and run echo with your name.
2. Run bash --version and note the version number.
3. Run echo $BASH_VERSION — this prints the version from inside Bash.
4. Type help to see a list of Bash built-in commands.
```

Common Mistake: Confusing Terminal and Shell

```
The terminal is the window. The shell (Bash) is the program running inside it. When people say
"open a terminal," they mean open the window AND run a shell inside it. You can change which
shell runs inside your terminal, but you cannot use a shell without some kind of terminal interface.
```

Project: Your First Game Output

```
Even a single echo command is game output. Create a file called hello.sh :


#!/bin/bash
echo "================================"
echo "   Welcome to Terminal Quest!   "
echo "================================"
echo ""
echo "Press Enter to begin..."
read
echo "Let the adventure begin!"



Run it with: bash hello.sh . This is the skeleton of every terminal game — display text, read input,
respond.
```

Chapter 2

Essential Linux Commands

What You Will Learn

The core Linux commands you will use constantly: listing files, moving around, creating and deleting

things, and getting help. These are the building blocks your scripts and games will call upon.

The 20 Commands You Must Know

Command	Purpose	Example
ls	List files	ls -la
cd	Change directory	cd /tmp
pwd	Print working directory	pwd
mkdir	Make directory	mkdir games
rmdir	Remove empty directory	rmdir olddir
touch	Create empty file	touch savefile.txt
cp	Copy file/directory	cp a.txt b.txt
mv	Move/rename	mv old.txt new.txt
rm	Remove file	rm junk.txt
cat	Display file contents	cat story.txt
less	Page through file	less logfile.txt
head	Show first lines	head -10 file.txt
tail	Show last lines	tail -5 file.txt
find	Find files	find . -name '*.sh'
grep	Search text	grep 'error' log.txt
wc	Count lines/words	wc -l file.txt
sort	Sort lines	sort scores.txt
chmod	Change permissions	chmod +x game.sh
man	Read manual	man ls
echo	Print text	echo Game Over

Listing Files with ls

```
# List files in current directory
ls

# Long format with permissions, size, date
ls -l

# Include hidden files (starting with .)
ls -a

# Human-readable sizes (KB, MB, GB)
ls -lh

# Sort by modification time (newest first)
ls -lt

# Reverse sort (newest last)
ls -ltr

# Combine flags
ls -lah




ls -l output explained:

-rwxr-xr-- 1 user group 2048 Jan 5 10:30 game.sh
|├─┤├─┤├─┤ |  |  |  
```

```
# Find all .sh files in current directory tree
find . -name '*.sh'

# Find all directories named 'save'
find /home -type d -name 'save'

# Find files modified in last 24 hours
find . -mtime -1

# Find files larger than 1MB
find . -size +1M

# Find and execute a command on each result
find . -name '*.tmp' -exec rm {} \;

# Find empty files
find . -empty
```

Tip: find vs ls -R

```
ls -R recursively lists everything, but find lets you filter. For games, find is perfect for
locating save files, configuration, or game assets in a directory structure.
```

The wc Command

Count lines, words, and characters — useful for score tracking and text processing.

```
wc file.txt        # lines words chars
wc -l file.txt     # just line count
wc -w file.txt     # just word count
wc -c file.txt     # just byte count
echo "hello world" | wc -w    # 2
```

sort and uniq

Sort lines and remove duplicates — perfect for high-score tables.

```
# Sort a high-score file numerically (descending)
sort -rn scores.txt

# Sort and remove duplicates
sort -u names.txt

# Show top 3 scores
sort -rn scores.txt | head -3
```

Exercise: Command Practice

```
1. Create a directory called bash_games and navigate into it.
2. Create 5 empty files: level1.txt through level5.txt .
3. List them sorted by name.
4. Use find to locate all .txt files.
5. Run man ls and read about the -S flag (sort by size).
```

Common Mistake: Using rm Carelessly

```
The rm command has no recycle bin. Once a file is deleted, it is gone. Common errors:

• rm *.txt — fine, but make sure you're in the right directory
• rm -rf / — catastrophic. Never run this.
• rm $var where $var is empty — becomes rm , which does nothing, or worse

Always use ls to preview before rm , and quote your variables.
```

Project: Score Board Creator

```
Build a simple script that creates and displays a score board:


#!/bin/bash
# scoreboard.sh — create and display a scoreboard

echo "Player1 9500" > scores.txt
echo "Player2 7200" >> scores.txt
echo "Player3 12000" >> scores.txt
echo "Player4 4500" >> scores.txt




echo "═══ HIGH SCORES ═══"
sort -rn -k2 scores.txt | head -5
echo "═══════════════════"
```

This creates a scores file, then sorts and displays the top 5. The -k2 flag tells sort to use the second field (the score number).

Chapter 3

Navigating the Filesystem

What You Will Learn

The Linux filesystem is a tree. You will learn about absolute and relative paths, special directory

shortcuts, and how to move around efficiently — essential knowledge for any script that reads or

writes game files.

The Directory Tree

```
/
├── home/
│   ├── user/
│   │   ├── games/
│   │   │   ├── quest.sh
│   │   │   └── saves/
│   │   ├── documents/
│   │   └── .bashrc
│   └── otheruser/
├── bin/         ← essential programs
├── etc/         ← system config
├── tmp/         ← temporary files
├── usr/         ← user programs
│   ├── bin/
│   └── share/
└── var/         ← variable data (logs, etc.)
```

Absolute vs Relative Paths

An absolute path starts from the root (/) and specifies every directory. A relative path starts from

your current directory.

```
# Absolute path — works from anywhere
cd /home/user/games

# Relative path — depends on where you are
cd games
cd -./documents
cd ./saves
```

Special Directory Shortcuts

Symbol	Meaning	Example
.	Current directory	cd . (no change)
-.	Parent directory	cd -.
~	Home directory	cd ~

-	Previous directory	cd -
/	Root directory	cd /

Understanding pwd

pwd (print working directory) tells you exactly where you are:

```
$ pwd
/home/user/games

# Use it in scripts to store the current location
SCRIPT_DIR=$(pwd)
echo "Running from: $SCRIPT_DIR
```

> Tip: The ~ Shortcut in Scripts

```
In scripts, ~ may not always expand correctly. Use $HOME instead, which is more reliable:


# Less reliable in scripts
cd ~/games

# More reliable
cd "$HOME/games"

# Store script directory for relative references
SCRIPT_DIR="$(cd "$(dirname "$0")" && pwd)"
echo "Script is located at: $SCRIPT_DIR
```

Creating and Removing Directories

```
# Create a single directory
mkdir saves

# Create nested directories at once




mkdir -p games/saves/backup

# Create multiple directories
mkdir level1 level2 level3

# Remove an empty directory
rmdir saves

# Remove a directory and everything inside
rm -r old_games

# Remove without prompting
rm -rf temp_build




mkdir -p games/saves/backup creates:

games/          ← created
└── saves/      ← created
└── backup/ ← created

All at once, even if 'games' didn't exist.
```

Path Manipulation Tools

Bash provides built-in path manipulation through parameter expansion (covered in depth in Chapter

14):

```
filepath="/home/user/games/quest.sh"

# Extract the directory
echo "${filepath%-*}"        # /home/user/games

# Extract the filename
echo "${filepath##-/}"        # quest.sh

# Extract the extension
echo "${filepath##*.}"         # sh

# Remove the extension
echo "${filepath%.*}"          # /home/user/games/quest
```

Exercise: Filesystem Navigation

```
1. From your home directory, create the path projects/bash_book/chapter03 .
2. Navigate into it using a relative path from home.
3. Print your current directory with pwd .



4. Go back to home using cd ~ .
5. Jump to your previous directory with cd - .
6. Create a file inside chapter03 from your home directory using an absolute path.
```

Common Mistake: Spaces in Paths

Directory and file names with spaces cause problems in Bash because spaces separate arguments. Always quote paths:

```
# This BREAKS — Bash sees two arguments
cd /home/user/my games

# This WORKS — quotes keep it as one path
cd "/home/user/my games"

# Better: use underscores or hyphens
mkdir my_games
mkdir my-games
```

Project: Save File Manager

Create a script that manages game save directories:

```
#!/bin/bash
# save_manager.sh — manage game save directories

SAVE_ROOT="$HOME/game_saves"

# Create the save directory structure
mkdir -p "$SAVE_ROOT/slot1"
mkdir -p "$SAVE_ROOT/slot2"
mkdir -p "$SAVE_ROOT/slot3"

echo "Save directory structure created at: $SAVE_ROOT"
echo ""
echo "Contents:"
ls -R "$SAVE_ROOT"
echo ""
echo "Current location: $(pwd)"
echo "Save root: $SAVE_ROOT"
```

Run it and verify the directory tree was created. This pattern — creating a well-organized directory structure — is the foundation of every game that saves progress.

Chapter 4

Working with Files and Directories

What You Will Learn

How to create, copy, move, rename, and delete files. You will also learn about file types, symbolic

links, and how to inspect file content — skills you will use for game assets, save files, and

configuration.

Creating Files

```
# Create an empty file
touch highscores.txt

# Create a file with content
echo "Level 1: Complete" > progress.txt

# Append to a file
echo "Level 2: Complete" >> progress.txt
echo "Level 3: Complete" >> progress.txt

# Create a file from heredoc (multi-line)
cat > intro.txt << 'EOF'
Welcome to Dungeon Quest!
You are a brave adventurer.
Your mission: find the golden key.
EOF



File operations:

>    ──►   Write (overwrite/create)
>>   ──►   Append (add to end)
<    ──►   Read from file (input)
2>   ──►   Redirect errors
&>   ──►   Redirect both stdout and stderr

┌─────────┐      >      ┌─────────┐
│ Command │ ──────────► │ file    │   (overwrite)
│ Output  │ ──► >> ──►  │ file    │   (append)
└─────────┘             └─────────┘
```

Copying Files

```
# Copy a file
cp save1.txt save1_backup.txt

# Copy and preserve attributes (timestamps, permissions)
cp -p config.txt config_backup.txt

# Copy a directory recursively
cp -r game_assets/ game_assets_backup/

# Copy interactively (prompt before overwrite)
cp -i important.txt backup.txt

# Copy multiple files to a directory
cp level1.txt level2.txt level3.txt levels/
```

Moving and Renaming

```
# Rename a file
mv old_name.txt new_name.txt

# Move a file to another directory
mv save.txt saves/

# Move multiple files
mv *.txt documents/

# Move and don't overwrite existing
mv -n source.txt destination.txt

# Prompt before overwriting
mv -i file.txt /tmp/file.txt
```

Tip: mv vs cp

```
mv is instantaneous — it just updates the directory entry. cp actually copies the data. For large
files, mv is much faster. Use mv when you want to relocate, cp when you want to duplicate.
```

Deleting Files

```
# Delete a single file
rm unwanted.txt

# Delete multiple files
rm level1.old level2.old level3.old




# Delete with confirmation
rm -i important.txt

# Delete recursively (directories + contents)
rm -r old_project/

# Force delete (no prompts, ignore nonexistent)
rm -f tempfile.tmp
```

Viewing File Contents

```
# Print entire file
cat story.txt

# Print with line numbers
cat -n story.txt

# View page by page (q to quit, / to search)
less bigfile.txt

# First 10 lines
head -10 logfile.txt

# Last 10 lines
tail -10 logfile.txt

# Follow a file as it grows (great for live game logs)
tail -f game.log

# View specific line range
sed -n '5,10p' file.txt
```

File Types

The file command identifies what kind of data a file contains:

```
$ file game.sh
game.sh: Bourne-Again shell script, ASCII text executable

$ file scores.dat
scores.dat: ASCII text

$ file image.png
image.png: PNG image data, 1920 x 1080, 8-bit/color RGB, non-interlaced

$ file /bin/ls
/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV)
```

Symbolic Links (Symlinks)

A symlink is a pointer to another file. Think of it as a shortcut.

```
# Create a symlink
ln -s /home/user/games/quest.sh /home/user/quest

# Verify it
ls -l quest
# lrwxrwxrwx 1 user user 27 Jan 5 10:00 quest >> /home/user/games/quest.sh

# Symlink a directory
ln -s /home/user/games /games_link

# Find broken symlinks
find . -type l ! -exec test -e {} \; -print



Symlinks vs Hard Links:

Symlink (soft link):             Hard Link:
quest ──► quest.sh (target)      file2 ──┐
(pointer to path)               file1 ──┼──► same inode
(same data)

- Survives target deletion: NO      - Survives target deletion: YES
- Can link directories: YES         - Can link directories: NO
- Can cross filesystems: YES        - Can cross filesystems: NO
```

Exercise: File Management

```
1. Create a file called game_config.txt with 5 lines of content.
2. Copy it to game_config_backup.txt .
3. Rename the backup to config.old .
4. View the file with cat -n to see line numbers.
5. Create a symlink to game_config.txt called config .
6. Delete config.old and verify the symlink still works.
```

Common Mistake: Overwriting Files with >

```
The > operator overwrites the file. The >> operator appends. Mixing them up destroys data:


# This DESTROYS existing scores:
echo "NewPlayer 100" > scores.txt



# This ADDS to existing scores:
echo "NewPlayer 100" >> scores.txt

# Safe pattern: check if file exists first
if [ -f scores.txt ]; then
echo "Player 100" >> scores.txt
else
echo "Player 100" > scores.txt
fi
```

Project: Game Log System

Create a script that logs game events to a file:

```
#!/bin/bash
# game_log.sh — log game events

LOGFILE="game.log"

# Create log file if it doesn't exist
if [ ! -f "$LOGFILE" ]; then
echo "=== Game Log Created: $(date) ===" > "$LOGFILE"
fi

# Log some events
echo "[$(date '+%H:%M:%S')] Player started game" >> "$LOGFILE"
echo "[$(date '+%H:%M:%S')] Player entered dungeon" >> "$LOGFILE"
echo "[$(date '+%H:%M:%S')] Player found treasure" >> "$LOGFILE"
echo "[$(date '+%H:%M:%S')] Player defeated by dragon" >> "$LOGFILE"
echo "[$(date '+%H:%M:%S')] Game over" >> "$LOGFILE"

# Display the log
echo "=== Game Log ==="
cat -n "$LOGFILE"
echo ""
echo "Last 3 events:"
tail -3 "$LOGFILE"
```

This demonstrates file creation, appending, and reading — the core operations for any game that

tracks state or history.

Chapter 5

Text Viewing and Processing

What You Will Learn

How to view, search, count, sort, and transform text files. These tools are the backbone of processing game data, dialogue files, maps, and configuration.

cat — Concatenate and Display

```
# Display a file
cat story.txt

# Display multiple files
cat chapter1.txt chapter2.txt chapter3.txt

# Display with line numbers
cat -n dialogue.txt

# Display with non-printing characters visible
cat -A map.txt

# Concatenate files into a new file
cat part1.txt part2.txt part3.txt > full_story.txt

# Create a file from keyboard input (Ctrl+D to end)
cat > newfile.txt
```

less — The Pager

less lets you scroll through long files one page at a time.

Key	Action
Space / Page Down	Next page
b / Page Up	Previous page
Arrow keys	Scroll line by line
/pattern	Search forward for pattern
?pattern	Search backward
n	Next search match

N	Previous search match
g	Go to beginning
G	Go to end
q	Quit

head and tail — View the Ends

```
# First 10 lines (default)
head file.txt

# First N lines
head -20 file.txt

# Last 10 lines (default)
tail file.txt

# Last N lines
tail -5 file.txt

# Follow a growing file (real-time)
tail -f game.log

# Show last 20 lines and follow
tail -n 20 -f game.log

# Everything except last 5 lines
head -n -5 file.txt

# Everything except first 3 lines
tail -n +4 file.txt
```

> Tip: tail -f for Live Game Logs

```
When developing a game, tail -f is invaluable. Open two terminals: one running the game, one
running tail -f game.log . You will see log entries appear in real time as the game runs —
perfect for debugging.
```

cut — Extract Columns

```
# Extract characters 1-5
cut -c1-5 dialogue.txt

# Extract field 2 using space delimiter
cut -d' ' -f2 scores.txt




# Extract fields 1 and 3 using comma delimiter (CSV)
cut -d',' -f1,3 players.csv

# Example: extract player names from scores file
echo "Alice 9500" | cut -d' ' -f1   # Alice
```

tr — Translate Characters

```
# Convert lowercase to uppercase
echo "hello" | tr 'a-z' 'A-Z'        # HELLO

# Convert uppercase to lowercase
echo "HELLO" | tr 'A-Z' 'a-z'        # hello

# Delete specific characters
echo "h e l l o" | tr -d ' '         # hello

# Squeeze repeated characters
echo "hello     world" | tr -s ' '   # hello world

# Replace newlines with spaces
cat lines.txt | tr '\n' ' '
```

sort — Sort Lines

```
# Alphabetical sort (default)
sort names.txt

# Numeric sort
sort -n scores.txt

# Reverse sort
sort -r names.txt

# Numeric reverse (highest first)
sort -rn scores.txt

# Sort by field (e.g., 2nd field)
sort -k2 scores.txt

# Sort by field numerically
sort -k2 -n scores.txt

# Remove duplicates while sorting
sort -u names.txt

# Sort by month name
sort -M months.txt
```

uniq — Remove Duplicates

uniq removes adjacent duplicate lines. Always pipe through sort first to group duplicates together.

```
# Remove duplicate lines (must sort first)
sort names.txt | uniq

# Count occurrences of each line
sort words.txt | uniq -c

# Show only duplicate lines
sort names.txt | uniq -d

# Show only unique lines (appear once)
sort names.txt | uniq -u
```

wc — Word Count

```
# Lines, words, bytes
wc file.txt

# Just line count
wc -l file.txt

# Just word count
wc -w file.txt

# Just character count
wc -c file.txt

# Count lines in output of another command
ls | wc -l          # count files in directory
grep "error" log.txt | wc -l # count error lines
```

tee — Write to File and Screen

tee sends output to both the screen and a file simultaneously.

```
# Display output AND save to file
echo "Game started" | tee game.log

# Append to file
echo "Level 2" | tee -a game.log
```

Write to multiple files

echo "Save point" | tee save1.log save2.log

```
tee acts as a T-junction:

┌──► screen (stdout)
Command │
output ─┤
└──► file (game.log)

Without tee:       command > file     (screen sees nothing)
With tee:          command | tee file (screen AND file both get output)
```

Exercise: Text Processing

```
1. Create a file with 10 player names and scores: Player Score per line.
2. Sort the file by score (highest first) and display top 3.
3. Count how many players are in the file using wc .
4. Extract just the player names using cut .
5. Convert all names to uppercase using tr .
6. Use tee to display the sorted list and save it to sorted_scores.txt .
```

Common Mistake: uniq Without sort

uniq only removes consecutive duplicate lines. If your file has:

```
Alice
Bob
Alice     ← NOT removed (not adjacent to first Alice)
```

Always pipe through sort first:

```
sort names.txt | uniq      # Now all duplicates are removed
```

Project: Dialogue System

Build a simple dialogue file processor that extracts and displays character lines:

```
#!/bin/bash
# dialogue.sh — process a dialogue file

# Create a dialogue file
cat > dialogue.txt << 'EOF'
King: Welcome, brave adventurer!
Player: I seek the golden key.
King: The key lies in the dragon's lair.
Player: I am not afraid.
King: Then go, and may fortune favor you.
NPC: I can sell you a sword for 50 gold.
Player: I will take it.
EOF

echo "═══ Full Dialogue ═══"
cat -n dialogue.txt

echo ""
echo "═══ King's Lines ═══"
grep "King:" dialogue.txt

echo ""
echo "═══ Player's Lines ═══"
grep "Player:" dialogue.txt

echo ""
echo "═══ Character Count ═══"
echo "King: $(grep -c "King:" dialogue.txt) lines"
echo "Player: $(grep -c "Player:" dialogue.txt) lines"

echo ""
echo "═══ All Characters ═══"
cut -d: -f1 dialogue.txt | sort -u
```

This script creates a dialogue file, then uses cat , grep , cut , and sort to extract and analyze it — exactly the kind of text processing a game engine needs.

Chapter 6

Permissions and Ownership

What You Will Learn

Linux file permissions control who can read, write, and execute files. Understanding permissions is

essential because your scripts must be executable to run, and your game save files need proper

access control.

The Permission System

Every file and directory has permissions for three categories of users:

Category	Symbol	Meaning
Owner	u	The user who owns the file
Group	g	Members of the file's group
Others	o	Everyone else
All	a	All of the above

And three permission types:

Permission	Symbol	For Files	For Directories
Read	r	View contents	List files
Write	w	Modify contents	Create/delete files
Execute	x	Run as program	Enter (cd into)

Reading Permission Strings

```
-rwxr-xr--     decomposed:

-   rwx   r-x   r--
|    |     |     |
|    |     |     └── Others: read only
|    |     └──────── Group: read + execute
|    └────────────── Owner: read + write + execute
└─────────────────── File type (- = file, d = dir)

Octal equivalents:
r = 4, w = 2, x = 1

rwx = 4+2+1 = 7
r-x = 4+0+1 = 5
r-- = 4+0+0 = 4

So -rwxr-xr-- = 754
```

chmod — Change Permissions

There are two ways to set permissions: symbolic and octal (numeric).

Symbolic Method

```
# Make a script executable for the owner
chmod u+x game.sh

# Make it executable for everyone
chmod a+x game.sh

# Remove write permission for others
chmod o-w config.txt

# Set exact permissions: owner=rwx, group=rx, others=rx
chmod u=rwx,g=rx,o=rx game.sh

# Add execute and remove write from group
chmod g+x-w data.txt
```

Octal (Numeric) Method

```
# 755 = rwxr-xr-x (typical for scripts/programs)
chmod 755 game.sh

# 644 = rw-r--r-- (typical for data files)
chmod 644 scores.txt




# 700 = rwx------ (private — owner only)
chmod 700 secret_saves/

# 600 = rw------- (private file — owner read/write only)
chmod 600 password.txt

# 777 = rwxrwxrwx (everyone can do everything — AVOID)
chmod 777 shared_dir/   # Dangerous!
```

Octal	Symbolic	Common Use
755	rwxr-xr-x	Executable scripts

644	rw-r--r--	Regular data files
600	rw-------	Private files (keys, passwords)
700	rwx------	Private directories
666	rw-rw-rw-	Shared writable files
777	rwxrwxrwx	Avoid — too permissive

chown — Change Ownership

```
# Change file owner
sudo chown alice game.sh

# Change owner and group
sudo chown alice:gamers game.sh

# Recursively change ownership of a directory
sudo chown -R alice:gamers /home/games/
```

Running Scripts

To run a Bash script directly (without typing bash before it), it must have execute permission:

```
# 1. Create the script
cat > game.sh << 'EOF'
#!/bin/bash
echo "Game running!"
EOF

# 2. Make it executable
chmod +x game.sh

# 3. Run it directly
./game.sh




# Alternative: run without execute permission
bash game.sh

# Alternative: run in current shell (affects your environment)
source game.sh
# or
. game.sh




Three ways to run a script:

1. ./game.sh           ──► New shell process, needs +x permission
2. bash game.sh         ──► New shell process, no +x needed
3. source game.sh       ──► Same shell process, runs in current environment
. game.sh                (variables, functions persist after script ends)
```

umask — Default Permissions

New files are created with default permissions determined by umask :

```
# View current umask
umask           # e.g., 0022

# Set umask for more restrictive defaults
umask 077       # new files: 600, new dirs: 700 (private)

# Set umask for shared environment
umask 002       # new files: 664, new dirs: 775 (group-writable)
```

Tip: Always chmod +x Your Game Scripts

```
After creating a new game script, you must make it executable before you can run it with ./
game.sh . If you see "Permission denied", run chmod +x game.sh and try again. This is the #1 issue
beginners encounter.
```

Exercise: Permission Practice

```
1. Create a script file and make it executable with chmod +x .
2. Verify the permissions changed with ls -l .
3. Set permissions to 755 using the octal method and verify.
4. Create a private file with 600 permissions.
5. Try to run a script without execute permission — observe the error.



6. Run the same script with bash scriptname instead.
```

Common Mistake: 777 Permissions

Setting permissions to 777 means everyone can read, write, and execute the file. This is a security risk. Always use the minimum permissions needed:

```
• Scripts: 755 (owner can edit, others can run)
• Data files: 644 (owner can edit, others can read)
• Private files: 600 (only owner can access)
```

Project: Game Installer Script

Create a script that sets up a game directory with proper permissions:

```
#!/bin/bash
# install_game.sh — set up game directory with proper permissions

GAME_DIR="$HOME/my_game"

# Create directory structure
mkdir -p "$GAME_DIR"
mkdir -p "$GAME_DIR/saves"
mkdir -p "$GAME_DIR/assets"

# Create the game script
cat > "$GAME_DIR/game.sh" << 'EOF'
#!/bin/bash
echo "Welcome to My Game!"
echo "Save directory: $HOME/my_game/saves"
EOF

# Create a default save file
echo '{"level": 1, "score": 0}' > "$GAME_DIR/saves/default.json"

# Set permissions
chmod 755 "$GAME_DIR/game.sh"        # script: executable
chmod 644 "$GAME_DIR/saves/default.json" # save: read/write owner
chmod 700 "$GAME_DIR/saves"           # saves dir: private

echo "Game installed at: $GAME_DIR"
echo ""
echo "Directory structure:"
ls -lR "$GAME_DIR"
echo ""
echo "To play: $GAME_DIR/game.sh"
```

This installer demonstrates real-world permission management for a game project — scripts are

executable, saves are private, and the structure is properly organized.

Chapter 7

Pipes and Redirection

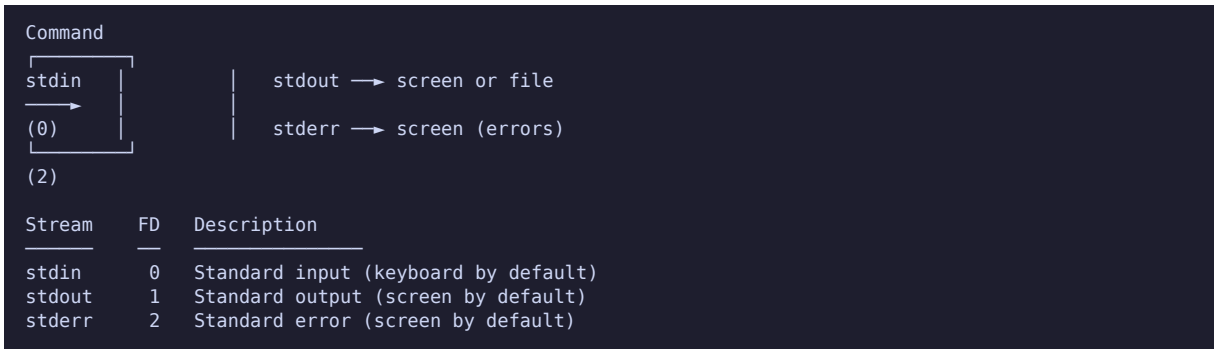
What You Will Learn

How to connect commands together with pipes, redirect input and output to files, and combine simple commands into powerful pipelines. This is one of the most important concepts in all of Unix

— and it directly mirrors how a game processes data through stages.

Standard Streams

Every command in Bash has three standard streams:


Command
stdin (0)
stdout → screen or file
stderr → screen (errors)
(2)
Stream FD Description
stdin 0 Standard input (keyboard by default)
stdout 1 Standard output (screen by default)
stderr 2 Standard error (screen by default)

Output Redirection

```
# Redirect stdout to a file (overwrite)
echo "Game Start" > game.log

# Append stdout to a file
echo "Level 1 complete" >> game.log

# Redirect stderr to a file
ls /nonexistent 2> errors.log

# Redirect both stdout and stderr to the same file
command &> output.log        # Bash 4+

# Alternative (more portable)
command > output.log 2>&1




# Redirect stdout to one file, stderr to another
command > results.txt 2> errors.txt

# Discard output (send to /dev/null)
command > /dev/null 2>&1



Redirection visualized:

echo "hi" > file        ──►   "hi" goes to file (overwrites)

echo "hi" >> file       ──►   "hi" appended to file

echo "hi" 2> errors     ──►   stdout goes to screen, stderr to file

echo "hi" > file 2>&1   ──►   both stdout and stderr go to file

echo "hi" > /dev/null   ──►   output goes to nowhere (discarded)

Order matters! 2>&1 must come AFTER the file redirect.
```

The Pipe (|)

The pipe connects the stdout of one command to the stdin of the next. This lets you chain commands into powerful pipelines.

```
# Count files in a directory
ls | wc -l

# Find all .sh files and sort them
find . -name '*.sh' | sort

# Search for "error" in a file, show first 5 matches
grep "error" logfile.txt | head -5

# Sort scores and show top 3
sort -rn scores.txt | head -3

# Chain multiple commands
cat players.txt | grep "active" | sort | head -10
```

```
Pipeline example: cat players.txt | grep "active" | sort | head -10

┌──────────────┐     ┌──────────────┐    ┌──────────────┐    ┌──────────┐
│ cat players  │──►  │ grep "active"│──► │     sort     │──► │ head -10 │
│     .txt     │     │              │    │              │    │          │
└──────────────┘     └──────────────┘    └──────────────┘    └──────────┘
reads file           filters lines      sorts them          shows top 10

Each command processes the output of the previous one.
Data flows left to right through the pipe.
```

Input Redirection

```
# Read input from a file instead of keyboard
sort < unsorted.txt

# Feed a file to a command
wc -l < bigfile.txt

# Use a here-string (single line)
grep "pattern" <<< "search this text"

# Heredoc (multi-line input)
cat << 'END'
Line 1
Line 2
Line 3
END
```

Command Substitution

Capture a command's output and use it as a value:

```
# Old style (backticks)
current_dir=`pwd`

# Modern style (preferred)
current_dir=$(pwd)

# Use in a sentence
echo "You are in: $(pwd)"

# Nested substitution
echo "There are $(ls | wc -l) files here"

# In game scripts




player_count=$(grep -c "player" savefile.txt)
echo "Players found: $player_count"
```

Tip: Always Use $() Instead of Backticks

```
The $() syntax is easier to read, nests naturally, and avoids confusion with quotation marks.
Backticks ( ` ) are legacy and harder to nest. Always prefer $(command) .
```

tee — Tee Junction

Capture output to a file and see it on screen simultaneously:

```
# Save output to file and display it
echo "Game started" | tee game.log

# Append mode
echo "Level 2" | tee -a game.log

# Save pipeline output
ls | tee filelist.txt | grep '\.sh$'

# Write to multiple files
echo "save" | tee save1.txt save2.txt save3.txt
```

Process Substitution

Advanced technique — treat a command's output as a temporary file:

```
# Compare two sorted files
diff <(sort file1.txt) <(sort file2.txt)

# Use a command output as input to a command expecting a file
grep "pattern" <(cat bigfile.txt)

# In game development: compare two save files
diff <(cat save1.json | sort) <(cat save2.json | sort)
```

Exercise: Pipes and Redirection

```
1. Use a pipe to count how many files are in /etc .
2. Redirect the output of ls -la to a file called listing.txt .
3. Append the current date to the same file using date and >> .



4. Sort a list of names and pipe through uniq to remove duplicates.
5. Use tee to display a message and save it to a file at the same time.
6. Use command substitution to store the number of files in a variable.
```

Common Mistake: Pipe Order Matters

Pipeline order is crucial. This works:

```
sort scores.txt | head -3       # sort first, then take top 3
```

This does NOT work the same way:

```
head -3 scores.txt | sort       # take first 3 lines, then sort THEM
```

The first gives you the top 3 scores. The second gives you the first 3 lines of the file, sorted alphabetically — completely different results.

Project: Game Data Pipeline

Build a pipeline that processes game combat logs:

```
#!/bin/bash
# combat_pipeline.sh — process combat data

# Create a combat log
cat > combat.log << 'EOF'
Player hit Goblin for 15 damage
Goblin hit Player for 8 damage
Player hit Goblin for 20 damage
Player hit Dragon for 12 damage
Dragon hit Player for 35 damage
Player hit Goblin for 15 damage
Goblin defeated
Player hit Dragon for 25 damage
Dragon defeated
EOF

echo "═══ Combat Analysis ═══"
echo ""

# Total combat lines
echo "Total events: $(wc -l < combat.log)"

# All hits by Player
echo ""




echo "Player attacks:"
grep "Player hit" combat.log

# Damage dealt to each enemy
echo ""
echo "Damage summary:"
grep "hit" combat.log | awk '{damage[$3]+=$5} END {for (e in damage) print e, damage[e]}' | sort | uniq -c | sort -rn

# Save analysis to file
echo ""
echo "Saving analysis to combat_report.txt..."
grep "defeated" combat.log | tee combat_report.txt

echo ""
echo "Report saved."
```

This pipeline uses grep , awk , sort , uniq , wc , tee , and command substitution together — a real-world example of how pipes combine simple tools into powerful data processors.

Chapter 8

Shell Variables and Environment

What You Will Learn

Variables store data your scripts and games need: player names, scores, health, inventory, game

state. You will learn how to create, read, modify, and export variables, and understand the difference

between shell and environment variables.

Creating and Using Variables

```
# Assign a variable — NO SPACES around =
player_name="Hero"
score=0
health=100

# Use a variable (prefix with $)
echo "Welcome, $player_name!"
echo "Score: $score, Health: $health"

# Brace syntax (preferred for clarity)
echo "Player: ${player_name}"
echo "File: ${player_name}_save.txt"

# Read input into a variable
echo "Enter your name:"
read name
echo "Hello, $name!"



Variable assignment rules:

CORRECT:                INCORRECT:
x="hello"               x = "hello"     ← spaces around = !
x=5                      x = 5           ← spaces around = !
x="$y"                   x ="hello"      ← space before =

The = must touch both the name and the value.
Bash treats spaces as argument separators.

Reading: $x or ${x} (braces avoid ambiguity)
${x}_suffix ← works (braces delimit the name)
$x_suffix     ← Bash looks for variable named x_suffix!
```

Quoting

Quoting controls how Bash interprets special characters. There are three types:

Quote Type	Behavior	When to Use
Double ("...")	Variables and $() expand; backslashes work	Most cases with variables

Single ('...)	Everything is literal — no expansion at all	Literal text, regex patterns
None	Variables expand, but word splitting and globbing occur	Simple values without spaces

```
name="World"

# Double quotes — variable expands
echo "Hello, $name"          # Hello, World

# Single quotes — literal, no expansion
echo 'Hello, $name'          # Hello, $name

# No quotes — variable expands, but risky with spaces
echo Hello, $name            # Hello, World

# Escape characters in double quotes
echo "He said \"hi\" to $name"    # He said "hi" to World

# Single quotes — cannot escape
echo 'Can\'t do this'       # Syntax error!
echo "Can't do this"         # This works
```

Tip: Always Quote Your Variables

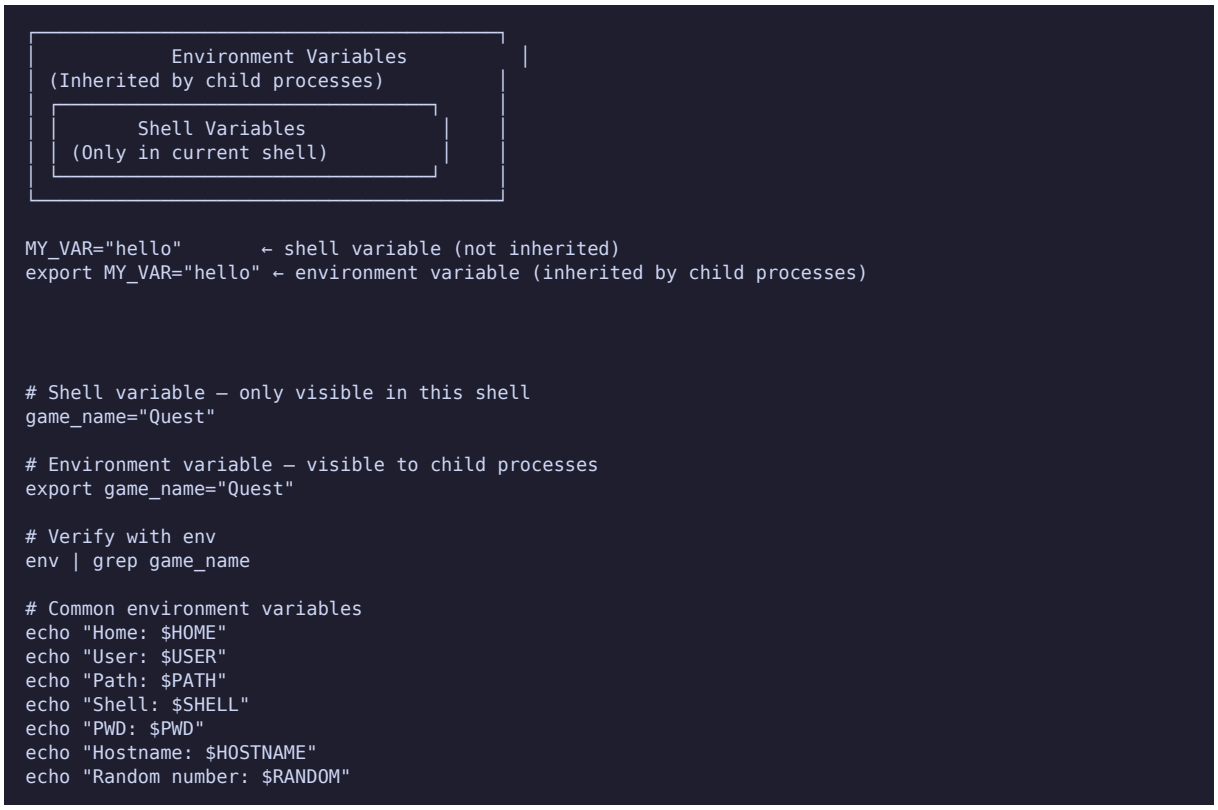
```
When in doubt, use double quotes around variables. This prevents word splitting and globbing
issues:


# Dangerous — if $file has spaces, this breaks
cp $file backup/

# Safe — quotes protect spaces
cp "$file" backup/

# Always use quotes in conditions
if [ "$var" = "yes" ]; then ...
```

Variable Scope: Shell vs Environment




```
MY_VAR="hello"       ← shell variable (not inherited)
export MY_VAR="hello" ← environment variable (inherited by child processes)
```

```
# Shell variable — only visible in this shell
game_name="Quest"

# Environment variable — visible to child processes
export game_name="Quest"

# Verify with env
env | grep game_name

# Common environment variables
echo "Home: $HOME"
echo "User: $USER"
echo "Path: $PATH"
echo "Shell: $SHELL"
echo "PWD: $PWD"
echo "Hostname: $HOSTNAME"
echo "Random number: $RANDOM"
```

Special Variables

Variable	Meaning
$0	Script name
$1 - $9	Positional parameters (arguments)
$#	Number of arguments
$@	All arguments (each quoted separately)
$*	All arguments (as single string)
$?	Exit status of last command
$$	Process ID (PID) of current shell

$!	PID of last background command
$RANDOM	Random number 0-32767
$LINENO	Current line number in script
$BASH_VERSION	Bash version string

#!/bin/bash

args.sh — demonstrate special variables

echo "Script name: $0"

echo "First arg: $1"

```
echo "Second arg: $2"
echo "Total args: $#"
echo "All args: $@"
echo "All args as string: $*"
echo "Exit status: $?"
echo "PID: $$"
echo "Random: $RANDOM"
# Run: ./args.sh apple banana cherry
```

Reading User Input

```
# Basic read
echo "Enter your name:"
read name
echo "Hello, $name"
# Read with a prompt (no echo needed)
read -p "Enter your name: " name
# Read silently (passwords)
read -s -p "Password: " password
echo ""
echo "Password length: ${#password}"
# Read with a timeout (seconds)
read -t 5 -p "Quick! Enter something (5s): " input
if [ $? -eq 0 ]; then
    echo "You entered: $input"
else
    echo "Too slow!"
fi
# Read a single character (no Enter needed)
read -n 1 -p "Press any key to continue... " key
echo ""
echo "You pressed: $key"
```

Integer Variables

Bash can treat variables as integers with declare -i :

```
declare -i health=100
declare -i damage=15

health=$((health - damage))
echo "Health: $health"      # 85

# Without declare -i, you need $(( ))
health=100
health=$((health - damage))
echo "Health: $health"      # 85
```

Read-only Variables

```
# Make a variable constant
readonly MAX_HEALTH=100
declare -r MAX_LEVEL=50

# Attempting to change causes an error
MAX_HEALTH=200    # bash: MAX_HEALTH: readonly variable
```

Exercise: Variables and Input

```
1. Create variables for a player: name, health (100), gold (50), level (1).
2. Print a character sheet using those variables.
3. Use read -p to let the user enter a new name.
4. Use $RANDOM to generate a random damage value (1-20).
5. Subtract the damage from health and display the result.
6. Export a variable and verify it with env .
```

Common Mistake: Unquoted Variables in Tests

```
Always quote variables in test conditions. If a variable is empty or contains spaces, unquoted use
causes syntax errors:


# BREAKS if $input is empty
if [ $input = "yes" ]; then ...      # becomes: [ = "yes" ] → error




# SAFE — quotes protect empty values
if [ "$input" = "yes" ]; then ... # becomes: [ "" = "yes" ] → OK
```

Project: Character Creation

Build an interactive character creation script:

```
#!/bin/bash
# character_create.sh — create a game character

echo "═══ Character Creation ═══"
echo ""

# Get player name
read -p "Enter character name: " char_name

# Choose class
echo "Available classes: Warrior, Mage, Rogue"
read -p "Choose your class: " char_class

# Set stats based on class
case "$char_class" in
[Ww]arrior)
health=150; attack=20; defense=15 ;;
[Mm]age)
health=80; attack=35; defense=5 ;;
[Rr]ogue)
health=100; attack=25; defense=10 ;;
*)
health=100; attack=15; defense=10 ;;
esac

# Generate random luck
luck=$((RANDOM % 10 + 1))

# Display character sheet
echo ""
echo "═══ Character Sheet ═══"
echo "Name:    $char_name"
echo "Class:   $char_class"
echo "Health: $health"
echo "Attack: $attack"
echo "Defense: $defense"
echo "Luck:    $luck/10"
echo "═══════════════════════"

# Save character
echo "$char_name,$char_class,$health,$attack,$defense,$luck" > character.save
echo "Character saved to character.save"
```

This script uses variables, input, conditionals, arithmetic, and file output — a complete character

creation system that could be the start of a real game.

Chapter 9

Shell Scripting Basics

What You Will Learn

How to write, structure, and execute Bash scripts. You will learn about shebangs, exit codes, comments, script organization, and the patterns that make scripts reliable and maintainable.

Your First Script

```
#!/bin/bash
# greeting.sh — a simple greeting script
# Usage: ./greeting.sh [name]

name="${1:-Adventurer}"    # Use first arg, or default to "Adventurer"
hour=$(date +%H)

if [ "$hour" -lt 12 ]; then
echo "Good morning, $name!"
elif [ "$hour" -lt 18 ]; then
echo "Good afternoon, $name!"
else
echo "Good evening, $name!"
fi

echo "Current time: $(date '+%H:%M:%S')"
echo "Have a great adventure!"
```

The Shebang Line

The first line of every script should be a shebang — it tells the system which interpreter to use:

```
#!/bin/bash           # Use the system Bash
#!/usr/bin/env bash    # Use Bash found via PATH (more portable)
#!/bin/sh              # Use the system POSIX shell
#!/usr/bin/python3     # Python 3 script



What happens when you run ./script.sh:

1. Kernel reads the shebang (#!)
2. Kernel launches the interpreter: /bin/bash
3. Bash reads the script file line by line
4. Each line is parsed and executed

Without the shebang, the system doesn't know how to run the file,
and you'll get an "Exec format error".
```

Making Scripts Executable

```
# Create the script
nano game.sh       # or vim, or any editor

# Make it executable
chmod +x game.sh

# Run it
./game.sh

# Or run without execute permission
bash game.sh
```

Exit Codes

Every command returns an exit code (0-255). 0 means success; anything else is an error. This is

how scripts communicate success/failure.

```
# Check exit status of a command
ls /etc > /dev/null 2>&1
echo "Exit code: $?"     # 0 if ls succeeded

ls /nonexistent 2>/dev/null
echo "Exit code: $?"     # non-zero (error)

# Set exit code in your script
exit 0    # success
exit 1    # general error
exit 2    # misuse of command

# Use exit codes in conditions
if grep -q "pattern" file.txt; then
echo "Found it!"
fi




# The && and || operators use exit codes
mkdir saves && echo "Created" || echo "Failed"
```

Exit Code	Meaning
0	Success
1	General error
2	Misuse of shell builtin
126	Command found but not executable
127	Command not found
128+N	Killed by signal N (e.g., 130 = SIGINT/Ctrl+C)

Script Arguments

```
#!/bin/bash
# args_demo.sh — demonstrate script arguments

echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"

# Check if arguments were provided
if [ $# -eq 0 ]; then
echo "Usage: $0 <player_name> <level>"
exit 1
fi

# Loop through all arguments
echo "Arguments loop:"
for arg in "$@"; do
echo " - $arg"
done

# Shift to consume arguments
shift
echo "After shift, \$1 is now: $1"
```

Comments

```
#!/bin/bash
# This is a full-line comment

echo "Hello"   # This is an inline comment




# ═══ Section Header ═══
# Use comments to organize your script into sections:
# - Configuration
# - Functions
# - Main logic
# - Cleanup

: 'This is a multi-line comment.
Bash does not have a true multi-line comment syntax.
This uses the : (no-op) builtin with a single-quoted string.
It is a common but unofficial pattern.'
```

Script Structure Template

Follow a consistent structure for all your scripts:

```
#!/bin/bash
#
# game.sh — Brief description of what this script does
#
# Usage: ./game.sh [options] [arguments]
#
# Author: Your Name
# Created: YYYY-MM-DD
#

# ── Configuration ──────────────────────────────────────
GAME_NAME="My Awesome Game"
VERSION="1.0"
MAX_PLAYERS=4

# ── Functions ──────────────────────────────────────────
show_help() {
echo "Usage: $0 [options]"
echo " -h     Show this help"
echo " -v     Show version"
}

# ── Main Logic ─────────────────────────────────────────

# Parse options
while getopts "hv" opt; do
case $opt in
h) show_help; exit 0 ;;
v) echo "$GAME_NAME v$VERSION"; exit 0 ;;
\?) echo "Unknown option: -$opt"; exit 1 ;;
esac
done

# Start the game
echo "Welcome to $GAME_NAME!"
# ... game logic here ...




# ── Cleanup ────────────────────────────────────────────
exit 0
```

set Options for Safer Scripts

```
#!/bin/bash
set -e    # Exit immediately if a command fails
set -u    # Treat unset variables as errors
set -o pipefail # Pipeline fails if any command fails
# Or combined:
set -euo pipefail

# Debug mode (print each command before executing)
set -x

# Turn off debug
set +x




set -e (errexit):
Command fails → script exits immediately
Prevents cascading errors

set -u (nounset):
Using undefined variable → script errors out
Prevents typos and logic errors

set -o pipefail:
pipe | cmd | pipe → fails if ANY part fails
(By default, only the LAST command's exit code matters)

set -x (xtrace):
Print each command as it executes
Essential for debugging
```

Tip: Start Every Script with set -euo pipefail

```
Adding set -euo pipefail at the top of your scripts catches errors early and prevents silent
failures. This is considered best practice in the Bash community.
```

Exercise: Script Writing

```
1. Write a script called info.sh that prints your username, home directory, and current date.
2. Add a shebang and make it executable.
3. Add a help message with -h flag.



4. Accept a name as an argument and greet that person.
5. Use a default value if no argument is given.
6. Add set -euo pipefail and verify the script still works.
```

Common Mistake: Forgetting the Shebang

Without #!/bin/bash as the first line, the system may try to run your script with sh (POSIX shell) instead of Bash. This causes errors when you use Bash-specific features like [[ ]] , arrays , or $RANDOM .

Project: Adventure Launcher

Create a script that launches different mini-games based on user choice:

```
#!/bin/bash
# launcher.sh — game launcher menu

set -euo pipefail

echo "═══ Game Launcher ═══"
echo "1. Guess the Number"
echo "2. Rock Paper Scissors"
echo "3. Dice Roller"
echo "4. Quit"
echo ""

read -p "Choose a game (1-4): " choice

case "$choice" in
1)
echo "Starting Guess the Number..."
target=$((RANDOM % 100 + 1))
attempts=0
while true; do
read -p "Guess (1-100): " guess
attempts=$((attempts + 1))
if [ "$guess" -lt "$target" ]; then
echo "Too low!"
elif [ "$guess" -gt "$target" ]; then
echo "Too high!"
else
echo "Correct! You won in $attempts tries!"
break
fi
done
;;
2)
echo "Starting Rock Paper Scissors..."
read -p "Rock, Paper, or Scissors? " player




choices=("rock" "paper" "scissors")
computer=${choices[$((RANDOM % 3))]}
echo "Computer chose: $computer"
if [ "$player" = "$computer" ]; then
echo "It's a tie!"
elif [ "$player" = "rock" ] && [ "$computer" = "scissors" ] || \
```

```
[ "$player" = "paper" ] && [ "$computer" = "rock" ] || \
[ "$player" = "scissors" ] && [ "$computer" = "paper" ]; then
echo "You win!"
else
echo "You lose!"
fi
;;
3)
echo "Rolling dice..."
die1=$((RANDOM % 6 + 1))
die2=$((RANDOM % 6 + 1))
echo "You rolled: $die1 and $die2 (total: $((die1 + die2)))"
;;
4)
echo "Goodbye!"
exit 0
;;
*)
echo "Invalid choice."
exit 1
;;
esac
```

This launcher uses variables, input, conditionals, loops, arrays, random numbers, and exit codes

— a complete game menu system. Save it, make it executable, and play.

Chapter 10

Conditionals and Logic

What You Will Learn

How to make decisions in your scripts. Conditionals let your game respond to player input, compare

values, validate data, and branch into different paths.

if Statements

```
# Basic if
if [ "$health" -le 0 ]; then
echo "Game Over!"
fi

# if-else
if [ "$score" -ge 1000 ]; then
echo "High score!"
else
echo "Keep trying!"
fi

# if-elif-else
if [ "$level" -lt 5 ]; then
echo "Beginner"
elif [ "$level" -lt 10 ]; then
echo "Intermediate"
elif [ "$level" -lt 20 ]; then
echo "Advanced"
else
echo "Master"
fi
```

test vs [ ] vs [[ ]]

Form	Type	Features
test	External command	Most portable, basic
[ ]	Same as test	Standard, widely portable
[[ ]]	Bash keyword	Pattern matching, regex, &&/||, no word splitting

```
# [ ] — standard POSIX test
if [ "$x" = "yes" ]; then ...
if [ "$num" -gt 10 ]; then ...
```

if [-f "file.txt"]; then ...

if ["$a" = "1"] && ["$b" = "2"]; then ...

[[ ]] — Bash enhanced test (preferred in Bash scripts)

if [["$x" == "yes"]]; then ...

if [["$num" -gt 10]]; then ...

if [[-f "file.txt"]]; then ...

if [["$a" == "1" && "$b" == "2"]]; then ... # && inside [[ ]]!

if [["$name" == D*]]; then ...	# pattern matching
if [["$input" =~ ^[0-9]+$]]; then ...	# regex matching

Tip: Prefer [[ ]] in Bash Scripts

```
Since this book focuses on Bash (not POSIX sh), always use [[ ]] instead of [ ] . It is more
powerful (supports regex, pattern matching, logical operators inside the brackets) and safer (no
word splitting or globbing on variables).
```

Numeric Comparisons

Operator	Meaning
-eq	Equal
-ne	Not equal
-lt	Less than
-le	Less than or equal
-gt	Greater than
-ge	Greater than or equal

if [[$score -gt $highscore]]; then

```
echo "New high score!"
```

fi

if [[$health -le 0]]; then

```
echo "You died!"
```

fi

Arithmetic comparison (alternative)

if ((score > highscore)); then

```
echo "New high score!"
```

fi

String Comparisons

Operator	Meaning
= or ==	Equal
!=	Not equal
<	Less than (alphabetical)
>	Greater than (alphabetical)
-z	String is empty

-n	String is not empty

```
if [[ "$name" == "admin" ]]; then
echo "Welcome, administrator!"
fi

if [[ -z "$input" ]]; then
echo "You didn't enter anything!"
fi

if [[ "$choice" == "quit" ]]; then
echo "Continuing..."
fi

# Pattern matching with [[ ]]
if [[ "$filename" == *.txt ]]; then
echo "Text file"
elif [[ "$filename" == *.sh ]]; then
echo "Shell script"
fi

# Regex matching with [[ ]] and =~
if [[ "$email" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
echo "Valid email"
fi
```

File Tests

Operator	True if...
-e	File exists
-f	File exists and is a regular file
-d	File exists and is a directory
-r	File is readable
-w	File is writable

-x	File is executable
-s	File exists and is not empty
-L	File is a symbolic link

```
# Check if save file exists
if [[ -f "save.json" ]]; then
echo "Loading saved game..."
source save.json
fi

# Check if directory exists
if [[ ! -d "saves" ]]; then
mkdir saves
fi

# Check if file is writable
if [[ -w "highscores.txt" ]]; then
echo "$player $score" >> highscores.txt
fi

# Check if file is empty
if [[ ! -s "progress.txt" ]]; then
echo "No progress found. Starting new game."
fi
```

Logical Operators

```
# AND (both must be true)
if [[ $health -gt 0 && $level -ge 5 ]]; then
echo "Strong enough to proceed!"
fi

# OR (either must be true)
if [[ "$class" == "mage" || "$class" == "wizard" ]]; then
echo "You can cast spells!"
fi

# NOT
if [[ ! -f "config.txt" ]]; then
echo "Config not found! Using defaults."
fi

# Short-circuit evaluation with && and ||
[[ -f save.txt ]] && echo "Save exists" || echo "No save"
mkdir -p saves && echo "Directory ready"
```

case Statements

The case statement is perfect for menu systems and pattern matching:

```
read -p "Choose: " choice

case "$choice" in
# Exact match
start)   echo "Starting game..." ;;

# Multiple patterns
quit|exit|q) echo "Goodbye!"; exit 0 ;;

# Pattern with wildcard
load*)   echo "Loading: ${choice#load }" ;;

# Character class
[Yy]*)   echo "Confirmed!" ;;
[Nn]*)   echo "Cancelled!" ;;

# Any number
[0-9]*) echo "You entered a number: $choice" ;;

# Default case (must be last)
*)       echo "Unknown command: $choice" ;;
esac




case statement flow:

┌──────────────┐
│ Read choice  │
└──────┬───────┘
       ▼
┌──────────────────────┐
│ pattern1) ... ;;     │── match ──► execute, exit case
└──────────────────────┘
       │ no match
       ▼
┌──────────────────────┐
│ pattern2) ... ;;     │── match ──► execute, exit case
└──────────────────────┘
       │ no match
       ▼
┌──────────────────────┐
│ *) default ... ;;    │── always matches
└──────────────────────┘
```

Arithmetic Conditionals

```
# Using (( )) for arithmetic
if (( health == 0 )); then
echo "You are dead!"
fi

if (( (score % 100) == 0 )); then
echo "Bonus level unlocked!"
fi

# Ternary-like expression
result=$(( score > highscore ? 1 : 0 ))
if (( result )); then
echo "New record!"
fi
```

Exercise: Conditionals

```
1. Write a script that checks if a file exists before deleting it.
2. Create a menu with at least 4 options using case .
3. Validate user input — check if they entered a number between 1 and 10.
4. Use [[ ]] with pattern matching to check if a filename ends in .txt .
5. Use && and || to write a one-line condition.
```

Common Mistake: Spaces in [ ] Tests

```
The [ ] and [[ ]] commands require spaces around the brackets and operators:


# WRONG — no spaces
if [$x=$y]; then ...    # Bash can't parse this
if [[ "$x"=="yes" ]]; then ... # Always true! Single non-empty token

# CORRECT — spaces around brackets and operators
if [ "$x" = "$y" ]; then ...
if [[ "$x" == "yes" ]]; then ...
```

Project: Battle System

```
Create a turn-based combat system using conditionals:
```

#!/bin/bash

battle.sh — simple combat system

player_hp=100

enemy_hp=80

enemy_name="Goblin"

echo "A wild $enemy_name appears!"

echo "Player HP: $player_hp | $enemy_name HP: $enemy_hp"

echo ""

while ((player_hp > 0 && enemy_hp > 0)); do

```
echo "── Actions ──"
echo "1. Attack"
echo "2. Defend"
echo "3. Heal"
read -p "Choose (1-3): " action

case "$action" in
1)
dmg=$((RANDOM % 20 + 10))
enemy_hp=$((enemy_hp - dmg))
echo "You hit $enemy_name for $dmg damage!"
;;
2)
echo "You brace for impact!"
block=1
;;
3)
heal=$((RANDOM % 15 + 10))
player_hp=$((player_hp + heal))
echo "You healed $heal HP!"
block=0
;;
*)
echo "Invalid action!"
block=0
continue
;;
esac

# Enemy turn
if (( enemy_hp > 0 )); then
enemy_dmg=$((RANDOM % 15 + 5))
if [[ "$block" == "1" ]]; then
enemy_dmg=$((enemy_dmg / 2))
echo "$enemy_name attacks for $enemy_dmg damage (blocked!)"
else
echo "$enemy_name attacks for $enemy_dmg damage!"
fi
player_hp=$((player_hp - enemy_dmg))
fi

echo ""
echo "Player HP: $player_hp | $enemy_name HP: $enemy_hp"
echo ""
block=0
```

done

```
if (( player_hp > 0 )); then
echo "═══ Victory! You defeated the $enemy_name! ═══"
else
echo "═══ Game Over! You were defeated! ═══"
fi
```

This battle system uses numeric comparisons, case statements, arithmetic evaluation, logical operators, file conditions, and loops — a complete turn-based combat loop that could be the core of a dungeon-crawler game.

Loops and Iteration

What You Will Learn

Loops repeat actions — processing items, game turns, animation frames, menu redraws. You will

learn for , while , until , break , continue , and how to use loops as game loops.

for Loops

```
# Iterate over a list
for fruit in apple banana cherry; do
echo "I like $fruit"
done

# Iterate over a range
for i in {1..5}; do
echo "Level $i"
done

# Range with step
for i in {0-.20-.2}; do     # 0, 2, 4, ..., 20
echo "Even: $i"
done

# Iterate over files
for file in *.txt; do
echo "Processing: $file"
done

# Iterate over command output
for line in $(cat players.txt); do
echo "Player: $line"
done

# C-style for loop
for ((i=0; i<5; i++)); do
echo "Iteration $i"
done

# Iterate over array elements
enemies=("goblin" "orc" "dragon")
for enemy in "${enemies[@]}"; do
echo "Enemy: $enemy"
done
```

while Loops

Basic while

count=0

while ((count < 5)); do

```
echo "Count: $count"
((count++))
```

done

Read lines from a file

while IFS= read -r line; do

```
    echo "Line: $line"
```

done < story.txt

Infinite loop (use break to exit)

while true; do

```
    read -p "Enter command (quit to exit): " cmd
    [[ "$cmd" == "quit" ]] && break
    echo "You said: $cmd"
```

done

While with condition

health=100

while ((health > 0)); do

```
    echo "HP: $health"
    health=$((health - 10))
```

done

echo "Game Over!"

```
Loop types at a glance:

for:   Know how many times to repeat
for i in 1 2 3 4 5; do ...; done

while: Repeat WHILE condition is true
while [[ $hp -gt 0 ]]; do ...; done

until: Repeat UNTIL condition is true
until [[ $ready == "yes" ]]; do ...; done

Game loop pattern:
while true; do
read input
process input
render output
[[ $game_over ]] && break
done
```

until Loops

until is the opposite of while — it loops until the condition becomes true:

```
# Wait until the player is ready
ready="no"
until [[ "$ready" == "yes" ]]; do
read -p "Are you ready? (yes/no): " ready
done
echo "Let's go!"

# Retry until success
until ping -c 1 google.com &>/dev/null; do
echo "Waiting for internet..."
sleep 2
done
echo "Connected!"
```

break and continue

```
# break — exit the loop immediately
for i in {1..10}; do
if (( i == 5 )); then
echo "Breaking at $i"
break
fi
echo "Number: $i"
done

# continue — skip to next iteration
for i in {1..10}; do
if (( i % 2 == 0 )); then
continue    # skip even numbers
fi
echo "Odd: $i"
done

# break with nested loops
for outer in 1 2 3; do
for inner in a b c; do
if [[ "$outer$inner" == "2b" ]]; then
break 2   # break out of BOTH loops
fi
echo "$outer$inner"
done
done
```

Reading Files Line by Line

```
# The correct way to read a file line by line
while IFS= read -r line; do
echo "Processing: $line"
done < input.txt

# Why IFS= and -r?
# IFS=    prevents leading/trailing whitespace trimming
# -r      prevents backslash interpretation
# Without these, lines with spaces or backslashes are mangled

# Read into multiple variables
while IFS-: read -r name class level; do
echo "Player: $name, Class: $class, Level: $level"
done < players.txt

# Read from a pipe
cat items.txt | while IFS= read -r item; do
echo "Item: $item"
done
# Note: variables set inside this loop are LOST after the loop!
# (because the pipe runs in a subshell)
```

Tip: The Subshell Pipe Problem

```
When you pipe into a while loop, the loop runs in a subshell. Variables modified inside the loop do
NOT persist after it ends. To work around this, use process substitution:


# BROKEN — count is lost after loop
count=0
cat file.txt | while read line; do
count=$((count + 1))
done
echo "Lines: $count"     # Still 0!

# WORKS — process substitution avoids subshell
count=0
while IFS= read -r line; do
count=$((count + 1))
done < <(cat file.txt)
echo "Lines: $count"     # Correct count!
```

select Menus

Bash has a built-in menu generator:

PS3="Choose an action: "

select action in "Attack" "Defend" "Heal" "Run" "Quit"; do

```
case "$action" in
"Attack") echo "You attack!" ;;
"Defend") echo "You defend!" ;;
"Heal")   echo "You heal!" ;;
"Run")    echo "You flee!" ;;
"Quit")   break ;;
*)        echo "Invalid choice." ;;
esac
```

done

Looping with seq

Generate a sequence

for i in $(seq 1 5); do

```
echo "Step $i"
```

done

With step

for i in $(seq 10 2 20); do	**# 10, 12, 14, 16, 18, 20**

```
echo "Value: $i"
```

done

Preferred: use brace expansion (no subshell)

for i in {1..5}; do

```
echo "Step $i"
```

done

Exercise: Loops

```
1. Write a for loop that prints numbers 1 to 10.
2. Use a while loop that counts down from 10 to "Blast off!"
3. Create a loop that processes each .txt file in a directory.
4. Write an infinite loop that breaks when the user types "quit".
5. Use continue to skip multiples of 3 in a 1-20 loop.
6. Build a select menu with 4 options.
```

Common Mistake: Forgetting to Update Loop Variables

```
Infinite loops happen when you forget to change the loop variable:


# INFINITE LOOP — count never changes!
count=0




while (( count < 5 )); do
echo "Count: $count"
# Missing: ((count++))
done

# Fix: always update the loop variable
count=0
while (( count < 5 )); do
echo "Count: $count"
((count++))
done
```

Project: Dungeon Explorer

Build a text-based dungeon explorer using loops as the game loop:

```
#!/bin/bash
# dungeon.sh — text-based dungeon explorer

player_hp=100
player_gold=0
room=1
max_rooms=10

echo "═══ Dungeon Explorer ═══"
echo "Navigate $max_rooms rooms. Find treasure. Survive!"
echo ""

while (( room < max_rooms && player_hp > 0 )); do
echo "─── Room $room of $max_rooms ───"

# Random event
event=$((RANDOM % 4))
case $event in
0)
# Treasure
gold=$((RANDOM % 50 + 10))
player_gold=$((player_gold + gold))
echo "You found $gold gold! (Total: $player_gold)"
;;
1)
# Trap
dmg=$((RANDOM % 20 + 5))
player_hp=$((player_hp - dmg))
echo "A trap! You take $dmg damage! (HP: $player_hp)"
;;
2)
# Healing potion
heal=$((RANDOM % 20 + 10))
player_hp=$((player_hp + heal))
echo "You found a healing potion! +$heal HP (HP: $player_hp)"
;;
3)
# Empty room



echo "The room is empty..."
;;
esac
```

```
(( room < max_rooms && player_hp > 0 )) && read -p "Press Enter to continue..."
echo ""
((room++))
done

echo ""
if (( player_hp > 0 )); then
echo "═══ You escaped the dungeon! ═══"
echo "Final HP: $player_hp"
echo "Gold collected: $player_gold"
else
echo "═══ You died in the dungeon... ═══"
echo "Gold collected: $player_gold"
fi
```

The entire game is built around a while loop that represents each dungeon room. Random events

provide variety, and the loop condition handles both victory (all rooms cleared) and defeat (HP reaches 0).

Chapter 12

Functions

What You Will Learn

Functions let you organize code into reusable blocks. In game development, functions are how you

separate combat logic, inventory management, rendering, and save/load systems into clean,

maintainable pieces.

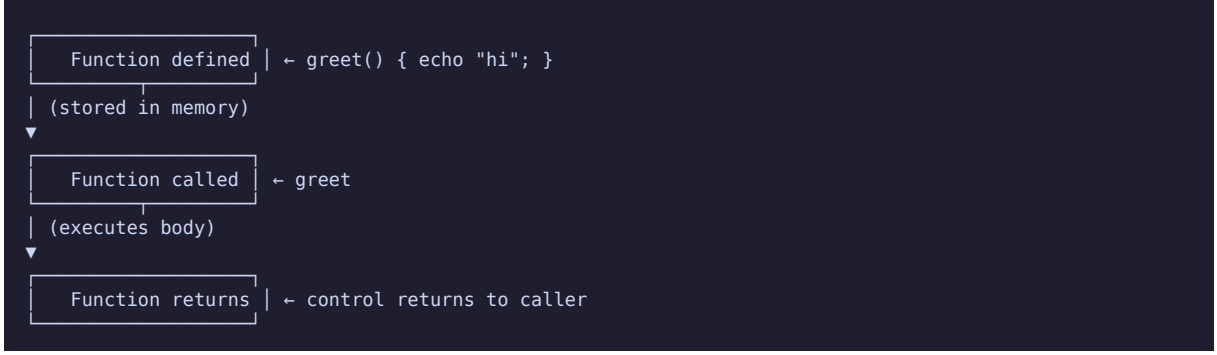
Defining and Calling Functions

```
# Basic function
greet() {
echo "Hello, adventurer!"
}

# Call it
greet

# Function with a name keyword
function greet {
echo "Hello, adventurer!"
}

# Call it
greet
```



Function Arguments

Functions receive arguments the same way scripts do — with $1 , $2 , $@ , etc:

```
# Function with arguments
greet_player() {
local name="$1"
local level="$2"
echo "Welcome $name! You are level $level."
}

# Call with arguments
greet_player "Alice" 5
greet_player "Bob" 12

# Function with variable arguments
sum_all() {
local total=0
for num in "$@"; do
total=$((total + num))
done
echo $total
}

sum_all 10 20 30      # 60
sum_all 1 2 3 4 5     # 15
```

The local Keyword

Without local , variables inside functions are global — they leak into the rest of the script. Always

use local for function-internal variables:

```
# BAD — variable is global
bad_function() {
counter=0          # This affects the global scope!
counter=$((counter + 1))
}

# GOOD — variable is local
good_function() {
local counter=0    # Only exists inside this function
counter=$((counter + 1))
echo $counter
}

# local is essential in game functions
calculate_damage() {
local base=$1
local modifier=$2
local damage=$((base * modifier / 100))




echo $damage
}
```

Tip: Always Use local Inside Functions

```
Forgetting local is one of the most common Bash bugs. Variables set without local persist after
the function returns, causing unexpected behavior. Always prefix function-internal variables with
local .
```

Return Values

Bash functions cannot return data directly. There are two patterns:

Pattern 1: Print and Capture

```
# Function prints a value, caller captures it
roll_dice() {
local sides=$1
echo $((RANDOM % sides + 1))
}

result=$(roll_dice 6)
echo "You rolled: $result"

result=$(roll_dice 20)
echo "D20 roll: $result"
```

Pattern 2: Return Exit Code

```
# Function returns 0 (success) or 1 (failure)
check_alive() {
local hp=$1
if (( hp > 0 )); then
return 0  # alive
else
return 1  # dead
fi
}

# Use with if
if check_alive $player_hp; then
echo "Still standing!"
else
echo "Game Over!"
fi
```

Pattern 3: Global Variable (advanced)

```
# Function sets a global variable
parse_input() {
local input="$1"
RESULT=""
case "$input" in
[0-9]*) RESULT="number" ;;
[a-zA-Z]*) RESULT="word" ;;
*) RESULT="unknown" ;;
esac
}

parse_input "hello"
echo "Type: $RESULT"    # word

parse_input "42"
echo "Type: $RESULT"    # number
```

Functions for Game Systems

```
#!/bin/bash
# game_functions.sh — reusable game functions

# ── Display Functions ──────────────────────────
draw_border() {
echo "╔══════════════════════════════╗"
echo "║ $1"
echo "╚══════════════════════════════╝"
}

draw_hp_bar() {
local current=$1
local max=$2
local bar_width=20
local filled=$((current * bar_width / max))
local empty=$((bar_width - filled))
local bar=""
for ((i=0; i<filled; i++)); do bar+="█"; done
for ((i=0; i<empty; i++)); do bar+="░"; done
echo "HP: [$bar] $current/$max"
}

# ── Game Logic Functions ───────────────────────
roll_dice() {
local sides=${1:-6}
echo $((RANDOM % sides + 1))
}

calculate_damage() {
local attack=$1
local defense=$2
local base=$((attack - defense))
(( base < 1 )) && base=1




local variance=$((RANDOM % 5 - 2))
echo $((base + variance))
```

}

── Save/Load Functions ──────────────────────────

save_game() {

```
local filename="save_${player_name}.txt"
echo "$player_name" > "$filename"
echo "$player_hp" >> "$filename"
echo "$player_gold" >> "$filename"
echo "$player_level" >> "$filename"
echo "Game saved to $filename"
```

}

load_game() {

```
    local filename="save_${1}.txt"
    if [[ ! -f "$filename" ]]; then
    echo "No save file found for $1"
    return 1
    fi
    # Read values
    player_name=$(sed -n '1p' "$filename")
    player_hp=$(sed -n '2p' "$filename")
    player_gold=$(sed -n '3p' "$filename")
    player_level=$(sed -n '4p' "$filename")
    echo "Welcome back, $player_name!"
```

}

── Usage ──

player_name="Hero"

player_hp=75

player_gold=200

player_level=5

draw_border " STATUS "

draw_hp_bar $player_hp 100

echo "Gold: $player_gold"

echo "Level: $player_level"

echo ""

dmg=$(calculate_damage 30 10)

echo "Damage dealt: $dmg"

save_game

Recursion in Bash

Factorial using recursion

factorial() {

```
    local n=$1
    if (( n == 1 )); then
    echo 1
    else
    local sub=$(factorial $((n - 1)))
    echo $((n * sub))
    fi
```

}

```
    echo "5! = $(factorial 5)"        # 120
    echo "10! = $(factorial 10)"     # 3628800
```

> Tip: Limit Recursion Depth

```
Bash is not optimized for deep recursion. The default stack limit allows a few hundred levels. For
game logic, prefer loops over recursion unless the problem is naturally recursive (like navigating a
tree structure).
```

Function Libraries

You can store functions in a separate file and source it:

```
# File: game_lib.sh
draw_border() { echo "╔═══════╗"; echo "║ $1"; echo "╚═══════╝"; }
roll_dice() { echo $((RANDOM % ${1:-6} + 1)); }

# File: game.sh
#!/bin/bash
source ./game_lib.sh     # Load functions

draw_border "Welcome!"
echo "You rolled: $(roll_dice 20)"
```

Exercise: Functions

```
1. Write a function that takes a name and prints a greeting.
2. Create a roll_dice function that accepts number of sides.
3. Write a function that returns the maximum of two numbers.
4. Create a draw_bar function that visualizes a percentage.
5. Build a function library in a separate file and source it.
6. Write a recursive function to count down from N to 1.
```

Common Mistake: Forgetting local in Functions

```
Variables without local are global. This causes hard-to-find bugs:


# BUG: i leaks into other loops
process_items() {
for i in 1 2 3; do    # i is GLOBAL




echo "Item $i"
done
}

# FIX: use local
process_items() {
local i               # i is LOCAL
for i in 1 2 3; do
echo "Item $i"
done
}
```

Project: Text RPG Engine

Build a modular text RPG with functions for each game system:

```
#!/bin/bash
# rpg.sh — modular text RPG engine

# ── Game State ──────────────────────────────────────
player_name="Hero"
player_hp=100
player_max_hp=100
player_gold=50
player_xp=0
player_level=1

# ── UI Functions ────────────────────────────────────
clear_screen() { clear; }

show_status() {
echo "┌──────────────────────────────┐"
echo "│ Name: $player_name (Lv $player_level)"
echo "│ HP:   $player_hp/$player_max_hp"
echo "│ Gold: $player_gold"
echo "│ XP:   $player_xp"
echo "└──────────────────────────────┘"
}

press_enter() {
echo ""
read -rp "Press Enter to continue..." _
}

# ── Combat Functions ────────────────────────────────
enemy_encounter() {
local enemy_name=$1
local enemy_hp=$2
local enemy_atk=$3

echo "An $enemy_name appears! (HP: $enemy_hp, ATK: $enemy_atk)"

while (( player_hp > 0 && enemy_hp > 0 )); do
show_status



echo "Enemy HP: $enemy_hp"
echo "1. Attack 2. Defend    3. Flee"
read -rp "> " choice
```

```
case "$choice" in
1)
local dmg=$((RANDOM % 15 + 10))
enemy_hp=$((enemy_hp - dmg))
echo "You deal $dmg damage!"
;;
2)
echo "You defend!"
local dmg=$((RANDOM % 10 + 5))
player_hp=$((player_hp - dmg / 2))
echo "Enemy deals $((dmg / 2)) damage (blocked!)"
continue
;;
3)
if (( RANDOM % 2 == 0 )); then
echo "You escaped!"
return 0
else
echo "Escape failed!"
fi
;;
esac

if (( enemy_hp > 0 )); then
local e_dmg=$((RANDOM % enemy_atk + 1))
player_hp=$((player_hp - e_dmg))
echo "$enemy_name deals $e_dmg damage!"
fi
done

if (( player_hp > 0 )); then
local xp_reward=$((enemy_hp * -1 + 20))
local gold_reward=$((RANDOM % 30 + 10))
player_gold=$((player_gold + gold_reward))
gain_xp $((RANDOM % 30 + 10))
echo "Victory! +$gold_reward gold"
return 0
else
echo "Defeat..."
return 1
fi
```

}

── Progression Functions ──────────────────────────────

gain_xp() {

```
local amount=$1
player_xp=$((player_xp + amount))
local needed=$((player_level * 100))
if (( player_xp >= needed )); then
player_level=$((player_level + 1))
player_max_hp=$((player_max_hp + 20))
player_hp=$player_max_hp
echo "LEVEL UP! You are now level $player_level!"
echo " Max HP increased to $player_max_hp!"
fi




}

# ── Shop Function ──────────────────────────────────────
visit_shop() {
echo "═══ Shop ═══"
echo "Gold: $player_gold"
echo "1. Health Potion (25g) 2. Leave"
read -rp "> " choice
case "$choice" in
1)
if (( player_gold >= 25 )); then
player_gold=$((player_gold - 25))
player_hp=$((player_hp + 30))
(( player_hp > player_max_hp )) && player_hp=$player_max_hp
echo "Bought! +30 HP"
else
echo "Not enough gold!"
fi
;;
esac
}

# ── Main Game Loop ─────────────────────────────────────
main() {
clear_screen
echo "═══ Welcome to Tiny RPG ═══"
press_enter

while (( player_hp > 0 )); do
clear_screen
show_status
```

```
echo ""
echo "1. Adventure 2. Shop 3. Quit"
read -rp "> " choice
case "$choice" in
1) clear; enemy_encounter "Goblin" 30 10; press_enter ;;
2) clear; visit_shop; press_enter ;;
3) echo "Goodbye!"; break ;;
esac
done

(( player_hp <= 0 )) && echo "═══ GAME OVER ═══"
}

main
```

This RPG engine organizes all game logic into functions: UI, combat, progression, shopping, and

the main loop. Each system is isolated and reusable — the foundation of good game architecture.

Arrays

What You Will Learn

Arrays store multiple values in a single variable. They are essential for game boards, inventories,

enemy lists, and any collection of related data. You will learn indexed arrays, associative arrays, and

how to use them as grids and boards.

Indexed Arrays

```
# Create an array
fruits=("apple" "banana" "cherry")

# Create an empty array
enemies=()

# Add elements
enemies+=("goblin")
enemies+=("orc" "dragon")

# Declare with specific elements
players=([0]="Alice" [1]="Bob" [2]="Carol")

# Access elements (0-indexed)
echo "${fruits[0]}"    # apple
echo "${fruits[1]}"    # banana
echo "${fruits[2]}"    # cherry

# Access all elements
echo "${fruits[@]}"      # apple banana cherry

# Number of elements
echo "${#fruits[@]}"     # 3

# Get indices
echo "${!fruits[@]}"     # 0 1 2
```

Array Operations

```
# Array length
arr=("a" "b" "c" "d")
echo "${#arr[@]}"              # 4

# Get specific element
echo "${arr[2]}"               # c




# Get a slice (from index 1, count 2)
echo "${arr[@]:1:2}"          # b c

# Append elements
arr+=("e" "f")
echo "${arr[@]}"              # a b c d e f

# Modify an element
arr[0]="AAA"
echo "${arr[@]}"              # AAA b c d e f

# Delete an element (leaves a gap)
unset 'arr[1]'
echo "${arr[@]}"              # AAA c d e f
echo "${!arr[@]}"            # 0 2 3 4 5

# Delete and re-index
arr=("${arr[@]}")
echo "${!arr[@]}"            # 0 1 2 3 4
```

Iterating Arrays

```
# Iterate over values
for fruit in "${fruits[@]}"; do
echo "Fruit: $fruit"
done

# Iterate with index
for i in "${!fruits[@]}"; do
echo "Index $i: ${fruits[$i]}"
done

# Iterate with C-style loop
for ((i=0; i-${#fruits[@]}; i++)); do
echo "[$i] = ${fruits[$i]}"
done
```

Reading Array from Input

```
# Read a line into an array
read -ra words <<< "hello world foo bar"
echo "${words[0]}"   # hello
echo "${words[1]}"   # world
echo "${#words[@]}" # 4

# Read file lines into array
mapfile -t lines < file.txt
echo "Line 1: ${lines[0]}"
echo "Total lines: ${#lines[@]}"




# Build array from command output
files=($(ls *.txt))




Indexed array visualization:

enemies=("goblin" "orc" "dragon")

Index:     0         1        2
┌──────────┬─────────┬──────────┐
│ goblin │ orc     │ dragon │
└──────────┴─────────┴──────────┘

${enemies[0]}     → goblin
${enemies[1]}     → orc
${enemies[2]}     → dragon
${enemies[@]}     → goblin orc dragon (all)
${#enemies[@]}    → 3 (count)
${!enemies[@]}   → 0 1 2 (indices)
```

Associative Arrays (Bash 4+)

Associative arrays use string keys instead of numeric indices — like dictionaries in Python or objects

in JavaScript.

```
# Declare an associative array
declare -A stats

# Assign key-value pairs
stats[name]="Hero"
stats[hp]=100
stats[mp]=50
stats[class]="Warrior"

# Access by key
echo "Name: ${stats[name]}"         # Hero
echo "HP: ${stats[hp]}"             # 100

# Iterate over values
for key in "${!stats[@]}"; do
echo "$key = ${stats[$key]}"
done

# Iterate over values directly
for val in "${stats[@]}"; do
echo "Value: $val"
done

# Check if key exists
if [[ -v stats[name] ]]; then
echo "Name is set!"




fi

# Count elements
echo "Total stats: ${#stats[@]}"    # 4



Associative array (key → value):

declare -A inventory

inventory["sword"]=1
inventory["potion"]=3
inventory["gold"]=250
```

```
┌──────────────┬──────────┐
│   Key        │ Value    │
├──────────────┼──────────┤
│ "sword"      │    1     │
│ "potion"     │    3     │
│ "gold"       │ 250      │
└──────────────┴──────────┘

${inventory[sword]} → 1
${inventory[gold]} → 250
```

2D Grids Using Arrays

Bash doesn't have true 2D arrays, but you can simulate them with naming conventions or index

arithmetic:

```
# Method 1: Named indices (board_row_col)
declare -A board
for r in 0 1 2; do
for c in 0 1 2; do
board[$r,$c]="."
done
done

# Place a piece
board[1,1]="X"
board[0,2]="O"

# Display the grid
for r in 0 1 2; do
row=""
for c in 0 1 2; do
row+="${board[$r,$c]} "
done
echo "$row"
done
# Output:
```

. . O

. X .

. . .

Game Board Example

#!/bin/bash

grid_game.sh — simple grid with player movement

ROWS=5

COLS=5

declare -A grid

Initialize grid

for ((r=0; r<ROWS; r++)); do

```
for ((c=0; c<COLS; c++)); do
grid[$r,$c]="·"
done
```

done

Place player and treasure

player_r=0; player_c=0

treasure_r=3; treasure_c=4

grid[$treasure_r,$treasure_c]="T"

render() {

```
clear
echo "Grid Game — find T (treasure)!"
echo "Controls: w=up, s=down, a=left, d=right, q=quit"
echo ""
for ((r=0; r<ROWS; r++)); do
row=""
for ((c=0; c<COLS; c++)); do
if [[ $r -eq $player_r && $c -eq $player_c ]]; then
row+=" @ "
else
row+=" ${grid[$r,$c]} "
fi
done
echo "$row"
done
```

}

while true; do

```
render
if [[ $player_r -eq $treasure_r && $player_c -eq $treasure_c ]]; then
echo ""
echo "You found the treasure! You win!"
break
fi
read -n 1 -rs key
case "$key" in
w) ((player_r > 0)) && ((player_r--)) ;;
s) ((player_r < ROWS-1)) && ((player_r++)) ;;
a) ((player_c > 0)) && ((player_c--)) ;;




d) ((player_c < COLS-1)) && ((player_c++)) ;;
q) echo "Quit!"; break ;;
esac
```

done

Exercise: Arrays

```
1. Create an array of 5 enemies and print each one.
2. Use an associative array to store player stats (name, hp, gold).
3. Write a function that adds an item to an inventory array.
4. Build a 3x3 grid using an associative array and display it.
5. Use mapfile to read a file into an array.
6. Sort an array of numbers using a loop.
```

Common Mistake: Forgetting Quotes Around ${{array[@]}}

Always quote {inline('"${{array[@]}}"')} when iterating. Without quotes, elements with spaces

break apart:

```
# BROKEN — "New York" splits into "New" and "York"
cities=("New York" "Los Angeles" "San Francisco")
for city in ${cities[@]}; do    # Missing quotes!
echo "$city"
done

# CORRECT — quotes preserve elements with spaces
for city in "${cities[@]}"; do
echo "$city"
done
```

Project: Inventory System

Build a game inventory system using associative arrays:

```
#!/bin/bash
# inventory.sh — game inventory system

declare -A inventory
declare -A item_descriptions

# Initialize items
inventory["sword"]=1
inventory["shield"]=1
```

inventory["potion"]=3

inventory["gold"]=150

inventory["key"]=0

item_descriptions["sword"]="A sharp steel blade"

item_descriptions["potion"]="Restores 30 HP"

item_descriptions["shield"]="Reduces damage by 50%"

item_descriptions["key"]="Opens locked doors"

item_descriptions["gold"]="Shiny currency"

add_item() {

```
local item=$1
local qty=${2:-1}
inventory[$item]=$((inventory[$item] + qty))
echo "Added $qty $item(s). Total: ${inventory[$item]}"
```

}

remove_item() {

```
local item=$1
local qty=${2:-1}
if [[ -v inventory[$item] ]] && (( inventory[$item] == qty )); then
inventory[$item]=$((inventory[$item] - qty))
(( inventory[$item] == 0 )) && unset 'inventory[$item]'
echo "Removed $qty $item(s)."
return 0
else
echo "Not enough $item!"
return 1
fi
```

}

show_inventory() {

```
echo "═══ Inventory ═══"
for item in "${!inventory[@]}"; do
local qty=${inventory[$item]}
local desc=${item_descriptions[$item]:-"Unknown item"}
printf " %-10s x%-3d %s\n" "$item" "$qty" "$desc"
done
echo "═══════════════════════"
```

}

has_item() {

```
local item=$1
local qty=${2:-1}
if [[ -v inventory[$item] ]] && (( inventory[$item] == qty )); then
return 0
else
return 1
fi
```

}

Test the inventory

show_inventory

echo ""

add_item "potion" 2

add_item "key"

echo ""

show_inventory

echo ""

```
remove_item "potion" 1
echo ""
if has_item "key"; then
echo "You can open the locked door!"
else
echo "You need a key."
fi
```

This inventory system demonstrates associative arrays for storage, functions for CRUD operations,

and practical game inventory management — a reusable module you can drop into any game.

Chapter 14

String Manipulation and Arithmetic

What You Will Learn

String manipulation and arithmetic are the heart of game logic. You will learn parameter expansion

for slicing, replacing, and transforming strings, plus arithmetic operations for damage calculation,

scoring, and random number generation.

Parameter Expansion Reference

Syntax	Description
${var}	Value of var
${#var}	Length of var
${var:offset}	Substring from offset
${var:offset:length}	Substring with length
${var#pattern}	Remove shortest prefix match
${var##pattern}	Remove longest prefix match
${var%pattern}	Remove shortest suffix match
${var%%pattern}	Remove longest suffix match
${var/old/new}	Replace first occurrence
${var-/old/new}	Replace all occurrences
${var^}	Uppercase first character
${var^^}	Uppercase all
${var,}	Lowercase first character
${var,,}	Lowercase all
${var:-default}	Use default if var is empty/unset
${var:=default}	Set var to default if empty/unset
${var:+alt}	Use alt if var is set/non-empty

String Length

```
name="Hello World"
echo "${#name}"              # 11

# Check if string is empty
if [[ -z "$input" ]]; then
echo "Empty string"
fi

# Compare lengths
if (( ${#password} < 8 )); then
echo "Password too short!"
fi
```

Substring Extraction

```
text="Hello, World!"

# From position 7
echo "${text:7}"             # World!

# From position 7, length 5
echo "${text:7:5}"          # World

# Negative offset (from end)
echo "${text: -6}"         # World! (note the space before -)
echo "${text: -6:5}"      # World

# Get first character
echo "${text:0:1}"           # H

# Get last character
echo "${text: -1}"           # !
```

Pattern Removal (Prefix/Suffix)

```
# # removes from the FRONT (prefix)
# % removes from the END (suffix)
# ## removes the LONGEST prefix match
# %% removes the LONGEST suffix match

file="/home/user/game/save.txt"

# Remove shortest prefix matching -/
echo "${file#-/}"          # home/user/game/save.txt

# Remove longest prefix matching -/
echo "${file##-/}"         # save.txt (just the filename)
```

Remove shortest suffix matching -*

echo "${file%-*}"	**# /home/user/game (directory path)**

Remove longest suffix matching .*

echo "${file%.*}"	**# /home/user/game/save (remove extension)**

Remove extension only

filename="quest.backup.tar.gz"

echo "${filename%%.*}"	**# quest**

echo "${filename%.*}"	# quest.backup.tar (first .gz removed)

```
Pattern removal cheat sheet:

${var#pattern}    ──►   remove from FRONT, shortest match
${var##pattern}   ──►   remove from FRONT, longest match
${var%pattern}    ──►   remove from END, shortest match
${var%%pattern}   ──►   remove from END, longest match

Mnemonic:
# is at the FRONT of $ on the keyboard → removes from front
% is at the END of $ on the keyboard → removes from end
## or %% → double = more aggressive (longest match)
```

String Replacement

text="The cat sat on the mat"

Replace first occurrence

echo "${text/cat/dog}"	**# The dog sat on the mat**

Replace all occurrences

echo "${text-/at/og}"	**# The cog sog on the mog**

Replace at the beginning (^ prefix)

echo "${text/#The/A}"	**# A cat sat on the mat**

Replace at the end (% suffix)

echo "${text/%mat/bed}"	**# The cat sat on the bed**

Delete (replace with nothing)

echo "${text-/ /_}"	**# The_cat_sat_on_the_mat**
echo "${text-/[^a-zA-Z]/_}"	# The_cat_sat_on_the_mat

Case Conversion (Bash 4+)

name="hello world"

Uppercase first letter

```
echo "${name^}"              # Hello world

# Uppercase all
echo "${name^^}"             # HELLO WORLD

# Lowercase first letter
echo "${name,}"              # hello world

# Lowercase all (already lowercase)
echo "${name,,}"             # hello world

# Uppercase specific characters
echo "${name^^[aeiou]}"      # hEllO wOrld
```

Default Values

```
# Use default if variable is empty or unset
echo "${player_name:-Anonymous}"   # Anonymous (if player_name is empty)

# Assign default AND use it
player_name="${player_name:=Hero}"   # Sets player_name to "Hero" if empty
echo "$player_name"

# Use alternate value if variable IS set
echo "${player_name:+(logged in)}" # (logged in) if player_name is non-empty
```

Arithmetic

Use $(( )) for arithmetic in Bash:

```
# Basic operations
echo $((5 + 3))        # 8
echo $((10 - 4))      # 6
echo $((6 * 7))       # 42
echo $((20 / 3))      # 6 (integer division!)
echo $((20 % 3))      # 2 (modulo/remainder)
echo $((2 ** 10))     # 1024 (exponent)

# Variables in arithmetic
hp=100
damage=15
hp=$((hp - damage))
echo "HP: $hp"        # 85

# Increment/decrement
count=0
((count++))           # count is now 1
((count++))           # count is now 2
((count--))           # count is now 1
((count += 10))       # count is now 11
((count -= 5))        # count is now 6
```

((count *= 2))	**# count is now 12**
((count -= 3))	# count is now 4

Comparison (returns 1 or 0)

echo $((5 > 3))	**# 1 (true)**
echo $((5 < 3))	# 0 (false)

Ternary operator

level=5

bonus=$((level > 3 ? 10 : 0))

echo "Bonus: $bonus" # 10

Random Numbers

$RANDOM generates 0-32767

echo $RANDOM

```
# Dice roll (1-6)
echo $((RANDOM % 6 + 1))
# Random in range (1-100)
echo $((RANDOM % 100 + 1))
# Random element from array
enemies=("goblin" "orc" "dragon" "skeleton")
random_enemy="${enemies[$((RANDOM % ${#enemies[@]}))]}"
echo "A $random_enemy appears!"
# Coin flip
if (( RANDOM % 2 == 0 )); then
    echo "Heads"
else
    echo "Tails"
fi
# Random percentage chance (25% chance)
if (( RANDOM % 100 < 25 )); then
    echo "Critical hit!"
fi
```

The (( )) Command

```
# (( )) for arithmetic evaluation
x=5
((x++))          # increment
((x *= 3))       # multiply
echo $x          # 18
# As a condition (no $ prefix needed inside)
if (( x > 10 )); then
    echo "Big number"
fi
# Multiple expressions
(( a = 5, b = 10, c = a + b ))
echo "$a $b $c"  # 5 10 15
```

String to Number Conversion

Strings are automatically treated as numbers in (( ))

num_str="42"

((num_str + 8))

echo $((num_str + 8))	**# 50**

But be careful with leading zeros (octal!)

echo $((010))	**# 8 (octal 10 = decimal 8)**
echo $((0x1F))	# 31 (hexadecimal)

Force base 10

echo $((10#010))	**# 10 (force decimal interpretation)**

Exercise: Strings and Arithmetic

```
1. Extract the filename from a full path using parameter expansion.
2. Replace all spaces in a string with underscores.
3. Convert a string to uppercase without using tr .
4. Roll 3d6 (three six-sided dice) and display the total.
5. Write a function that generates a random string of N characters.
6. Use the ternary operator to determine if a score is passing (>= 60).
```

Common Mistake: Integer Division in Bash

```
Bash arithmetic only works with integers. There is no floating-point math:


echo $((10 / 3))     # 3 (not 3.333...)
echo $((1 / 2))      # 0 (not 0.5)

# For floating point, use bc or awk
echo "scale=2; 10/3" | bc      # 3.33
echo "10 3" | awk '{print $1/$2}' # 3.33333

# For percentages, multiply first to avoid truncation
percentage=$((score * 100 / total))
```

Project: Combat Damage Calculator

Build a damage calculation system using string and arithmetic operations:

```
#!/bin/bash
# damage_calc.sh — RPG damage calculator

# Player stats
player_name="Aria"
player_str=25
player_dex=18
player_int=15
player_level=12

# Enemy stats
enemy_name="Ancient Dragon"
enemy_defense=30
enemy_hp=500

# ── Damage Functions ──────────────────────────────

# Physical attack: (STR * 2 + Level) - DEF, ±20% variance
calc_physical() {
local base=$((player_str * 2 + player_level))
local after_def=$((base - enemy_defense))
(( after_def < 1 )) && after_def=1
local variance=$((RANDOM % 41 - 20)) # -20 to +20
local damage=$((after_def * (100 + variance) / 100))
echo $damage
}

# Critical hit: 15% chance, 2x damage
apply_critical() {
local damage=$1
if (( RANDOM % 100 < 15 )); then
echo "crit $((damage * 2))"
else
echo "normal $damage"
fi
}

# Format damage string
format_damage() {
local type=$1
local amount=$2
local bar_len=$((amount / 10))
(( bar_len > 50 )) && bar_len=50
local bar=""
for ((i=0; i<bar_len; i++)); do bar+="█"; done
```

```
printf "%-8s %4d %s\n" "$type" "$amount" "$bar"
}

# ── Combat Simulation ──────────────────────────────
echo "═══ Combat Simulator ═══"
echo "Attacker: $player_name (STR:$player_str)"
echo "Defender: $enemy_name (DEF:$enemy_defense, HP:$enemy_hp)"
echo ""



total_damage=0
for ((turn=1; turn==5; turn++)); do
raw=$(calc_physical)
result=$(apply_critical "$raw")
type="${result%% *}"
amount="${result##* }"

echo "Turn $turn:"
format_damage "$type" "$amount"

if [[ "$type" == "crit" ]]; then
echo " CRITICAL HIT!"
fi

enemy_hp=$((enemy_hp - amount))
total_damage=$((total_damage + amount))

if (( enemy_hp <= 0 )); then
echo " DEFEATED: $enemy_name defeated!"
break
fi
echo " $enemy_name HP: $enemy_hp"
echo ""
done

echo "═══ Summary ═══"
echo "Total damage: $total_damage"
avg=$((total_damage / 5))
echo "Average per turn: $avg"
echo "Enemy remaining HP: $((enemy_hp < 0 ? 0 : enemy_hp))"
```

This calculator uses parameter expansion, arithmetic operations, random numbers, string formatting, and loops — a complete damage system that could power any RPG combat engine.

Chapter 15

Regular Expressions

What You Will Learn

Regular expressions (regex) are patterns for matching text. In game development, you use them for

input validation, parsing save files, filtering data, and implementing text-based commands. You will

learn regex syntax and how to use it with [[=~]] , grep , sed , and awk .

Regex Building Blocks

[abc] [^abc] [a-z] ^ \d \w \s (…) |

Pattern	Matches	Example
.	Any single character	c.t → cat, cot, cut
*	Zero or more of previous	ab* → a, ab, abbb
+	One or more of previous	ab+ → ab, abbb
?	Zero or one of previous	ab? → a, ab
{n}	Exactly n of previous	a{3} → aaa
{n,m}	Between n and m of previous	a{2,4} → aa, aaa, aaaa
Any one of a, b, c	[aeiou]	
NOT a, b, or c	[^0-9]	
Any lowercase letter	[a-z]+	
Start of string/line	^Hello	
$	End of string/line	world$
Digit (in some tools)	\d+	
Word character (in some tools)	\w+	
Whitespace (in some tools)	[[:space:]]+	
Capture group	(ab)+	
OR (alternation)	cat|dog	

```
Regex anatomy: ^[A-Z][a-z]+[0-9]{2,3}$

^          ── Start of string
[A-Z]      ── One uppercase letter
[a-z]+     ── One or more lowercase letters
[0-9]{2,3}   ── Two or three digits
$          ── End of string

Matches: "Ab42", "Zebra123", "Xy999"
Does NOT match: "ab42" (lowercase start), "A1" (needs 2+ digits after)
```

Regex in Bash with [[=~]]

Use [[=~]] to match regex patterns in Bash:

```
# Check if input is a number
input="12345"
if [[ "$input" =~ ^[0-9]+$ ]]; then
echo "Valid number"
fi

# Validate email format
email="user@example.com"
if [[ "$email" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
echo "Valid email"
fi

# Extract matched groups
phone="Call me at (555) 123-4567"
if [[ "$phone" =~ \(([0-9]{3})\) ([0-9]{3})-([0-9]{4}) ]]; then
echo "Area code: ${BASH_REMATCH[1]}"
echo "Prefix: ${BASH_REMATCH[2]}"
echo "Line: ${BASH_REMATCH[3]}"
fi

# Check for valid command format
command="move north"
if [[ "$command" =~ ^(move|go)[[:space:]]+(north|south|east|west)$ ]]; then
echo "Valid movement command"
fi
```

Tip: BASH_REMATCH Array

```
When a regex matches, Bash stores the results in BASH_REMATCH : $-{BASH_REMATCH[0]-} is the
full match, and $-{BASH_REMATCH[1]-} , $-{BASH_REMATCH[2]-} , etc. are capture groups. Use
these to extract data from matched patterns.
```

Common Regex Patterns for Games

```
# Player name (letters only, 3-15 chars)
name_regex='^[a-zA-Z]{3,15}$'

# Positive integer
int_regex='^[0-9]+$'

# Decimal number
float_regex='^[0-9]+\.[0-9]{1,2}$'

# Yes/No (any case)
yesno_regex='^[Yy]es$|^[Nn]o$|^[Yy]$|^[Nn]$'

# Menu choice (single digit 1-9)
menu_regex='^[1-9]$'

# Coordinates like "A5", "C3"
coord_regex='^[A-Z][0-9]+$'

# Dice notation like "3d6", "1d20", "2d4+2"
dice_regex='^[0-9]+d[0-9]+([+-][0-9]+)?$'
```

Regex with grep, sed, and awk

Each tool uses regex slightly differently:

```
# grep: Basic regex by default, -E for extended
grep 'pattern' file.txt         # basic regex
grep -E 'pattern' file.txt     # extended regex (same as egrep)
grep -P 'pattern' file.txt     # Perl-compatible regex (PCRE)

# sed: Uses basic regex, -E for extended
sed 's/pattern/replacement/' file.txt
sed -E 's/pattern/replacement/' file.txt

# awk: Uses extended regex (ERE)
awk '/pattern/' file.txt
awk '$1 ~ /pattern/' file.txt    # match against field 1
```

Feature	Basic (BRE)	Extended (ERE)
+, ?	Need backslash: \+ , \?	Direct: + , ?
|	Not available	Direct: |
()	Need backslash: \(\)	Direct: ()
{}	Need backslash: \-{\-}	Direct: {}

Tip: Use -E for Cleaner Regex

```
When using grep or sed, always pass -E (or use egrep ) to enable extended regex. It lets you use
+ , ? , | , and () without backslashes, making patterns much more readable.
```

Anchors and Boundaries

```
# ^ matches start of line
grep '^Error' log.txt         # Lines starting with "Error"

# $ matches end of line
grep 'done$' tasks.txt        # Lines ending with "done"

# Both: exact match
grep '^Complete$' status.txt # Lines that are exactly "Complete"

# Word boundary (\b in some tools)
grep -E '\bcat\b' animals.txt # "cat" but not "catch" or "scattered"
```

Quantifiers in Depth

```
# Greedy (default) — matches as much as possible
echo "<div>hello</div>" | grep -oE '<.+>'    # <div>hello</div>

# Lazy (in PCRE with -P) — matches as little as possible
echo "<div>hello</div>" | grep -oP '<.*?>'   # <div> then </div>

# Specific counts
echo "aaaaa" | grep -oE 'a{3}'    # aaa
echo "aaaaa" | grep -oE 'a{2,4}' # aaaa
echo "aaaaa" | grep -oE 'a{2,}' # aaaaa
echo "aaaaa" | grep -oE 'a{,3}' # aaa
```

Exercise: Regular Expressions

```
1. Write a regex to validate a username (alphanumeric, 3-12 chars).
2. Use [[ =~ ]] to check if input is a valid dice roll (like "2d6+3").
3. Write a regex to extract a number from "Score: 1234".
4. Use grep -E to find all email addresses in a text file.
5. Write a regex that matches "yes" in any case (YES, Yes, yes, y, Y).
6. Create a regex for map coordinates (letter+number, like B7, H3).
```

Common Mistake: Regex vs Glob Confusion

Bash has two different pattern systems. Do not confuse them:

```
# GLOB patterns (used in case, [[ == ]], and filename expansion)
*   = match anything
?   = match one character
[abc] = match one of a, b, c

if [[ "$file" == *.txt ]]; then ...   # GLOB match

# REGEX (used with [[ =~ ]])
.* = match anything
.   = match one character
[abc] = match one of a, b, c
+   = one or more

if [[ "$file" =~ .*\.txt$ ]]; then ...   # REGEX match

# They look similar but work differently!
# In globs: * means "anything"
# In regex: * means zero or more of previous
```

Project: Command Parser

Build a game command parser using regex:

```
#!/bin/bash
# command_parser.sh — parse player commands with regex

parse_command() {
local input="$1"

# Trim leading/trailing whitespace
input="${input## }"
input="${input%% }"

# Empty input
[[ -z "$input" ]] && echo "Empty command." && return

# Movement: go/move/walk + direction
if [[ "$input" =~ ^(go|move|walk)[[:space:]]+(north|south|east|west|up|down)$ ]]; then
echo "Moving ${BASH_REMATCH[2]}"
return
fi

# Short movement: just a direction
if [[ "$input" =~ ^(n|s|e|w|u|d)$ ]]; then
local dir
case "${BASH_REMATCH[1]}" in
n) dir="north" ;; s) dir="south" ;;




e) dir="east"     ;; w) dir="west"   ;;
u) dir="up"       ;; d) dir="down"   ;;
esac
echo "Moving $dir"
return
fi

# Take item: take/get/pick + item name
if [[ "$input" =~ ^(take|get|pick[[:space:]]+up)[[:space:]]+(.+)$ ]]; then
echo "Taking: ${BASH_REMATCH[2]}"
return
fi

# Attack: attack/fight + target
if [[ "$input" =~ ^(attack|fight|hit)[[:space:]]+(.+)$ ]]; then
echo "Attacking: ${BASH_REMATCH[2]}"
return
```

```
fi

# Look: look at/examine + target
if [[ "$input" =~ ^(look[[:space:]]+at|examine|inspect)[[:space:]]+(.+)$ ]]; then
echo "Examining: ${BASH_REMATCH[2]}"
return
fi

# Inventory check
if [[ "$input" =~ ^(inventory|inv|i|items)$ ]]; then
echo "Opening inventory..."
return
fi

# Help
if [[ "$input" =~ ^(help|h|\?)$ ]]; then
echo "Available: move <dir>, take <item>, attack <target>, look at <target>,
```

inventory, help"

```
return
fi

# Dice roll: NdM or NdM+K
if [[ "$input" =~ ^([0-9]+)d([0-9]+)([+-][0-9]+)?$ ]]; then
local num=${BASH_REMATCH[1]}
local sides=${BASH_REMATCH[2]}
local mod=${BASH_REMATCH[3]:-0}
local total=0
for ((i=0; i<num; i++)); do
total=$((total + RANDOM % sides + 1))
done
total=$((total + mod))
echo "Rolled ${num}d${sides}${BASH_REMATCH[3]:-""} = $total"
return
fi

# Quit
if [[ "$input" =~ ^(quit|exit|q)$ ]]; then
echo "Goodbye!"
exit 0
fi

echo "I don't understand '$input'. Type 'help' for commands."




}

# Test the parser
echo "═══ Command Parser Demo ═══"
echo "Type commands (or 'quit'):"
echo ""

while true; do
read -r -p "> " cmd
parse_command "$cmd"
echo ""
done
```

This parser uses regex to match and extract data from natural language commands. It handles movement, item interaction, combat, dice rolling, and help — the command system for a text adventure game.

grep — Text Search

What You Will Learn

grep is the ultimate text search tool. You will learn to find patterns in files, filter command output,

validate input, and build game features like dialogue search and log analysis.

Basic grep Usage

```
# Search for a pattern in a file
grep "error" logfile.txt

# Case-insensitive search
grep -i "error" logfile.txt

# Search in multiple files
grep "pattern" file1.txt file2.txt

# Recursive search in a directory
grep -r "pattern" /path/to/dir/

# Show line numbers
grep -n "pattern" file.txt

# Count matches
grep -c "pattern" file.txt

# Invert match (lines that DON'T match)
grep -v "pattern" file.txt

# Show only the matching part
grep -o "pattern" file.txt

# Extended regex
grep -E "pattern1|pattern2" file.txt

# Color highlight matches
grep --color=auto "pattern" file.txt
```

Common grep Flags

Flag	Long Form	Description
-i	--ignore-case	Case-insensitive
-v	--invert-match	Non-matching lines

-n	--line-number	Show line numbers
-c	--count	Count matches
-o	--only-matching	Only the matched text
-r	--recursive	Search directories
-l	--files-with-matches	Filenames only

-E	--extended-regex	Extended regex
-w	--word-regexp	Whole word match
-x	--line-regexp	Whole line match
-A N	--after-context=N	N lines after match
-B N	--before-context=N	N lines before match
-C N	--context=N	N lines around match
-q	--quiet	No output (exit code only)

Using grep in Pipelines

```
# Filter process list
ps aux | grep "bash"

# Find large files in a directory listing
ls -lh | grep -E "[0-9]+M"

# Count files by extension
ls | grep -c "\.txt$"

# Find running game processes
ps aux | grep -i "game" | grep -v grep

# Multiple patterns (OR)
grep -E "error|warning|critical" log.txt

# Multiple patterns (AND — pipe two greps)
grep "error" log.txt | grep "network"

# Exclude a pattern
grep "error" log.txt | grep -v "timeout"
```

Tip: grep -v grep Trick

```
When searching for a process with ps aux | grep process , the grep command itself appears in
results. Filter it out:



ps aux | grep "game" | grep -v grep     # Excludes the grep line itself
```

grep for Input Validation

```
# Check if input is numeric
echo "12345" | grep -qE '^[0-9]+$' && echo "Number!" || echo "Not a number!"

# Validate in a script
read -p "Enter age: " age
if echo "$age" | grep -qE '^[0-9]{1,3}$'; then
echo "Valid age: $age"
else
echo "Invalid input!"
fi

# Check if a word exists in a word list
if grep -qx "dragon" words.txt; then
echo "Valid word"
fi
# -x means the ENTIRE line must match
```

grep with Context

```
# Show 2 lines after each match
grep -A 2 "error" log.txt

# Show 2 lines before each match
grep -B 2 "error" log.txt

# Show 2 lines before AND after
grep -C 2 "error" log.txt

# Example: show the line around "GAME OVER"
grep -C 3 "GAME OVER" game_log.txt
```

Practical grep Patterns

```
# Find all TODO comments in your scripts
grep -rn "TODO" *.sh

# Find all function definitions
grep -E '^[a-z_]+\(\)' script.sh

# Find all variable assignments
grep -E '^[a-zA-Z_]+=' config.sh
```

Find empty lines

grep -E '^$' file.txt | wc -l

Find lines longer than 80 characters

grep -E '.{81,}' file.txt

Find IP addresses

grep -Eo '([0-9]{1,3}\.){3}[0-9]{1,3}' access.log

Find words containing specific letters

grep -E '^[a-z]*dragon[a-z]*$' words.txt

grep Exit Codes

grep returns 0 if match found, 1 if no match, 2 on error

grep -q "pattern" file.txt

if [$? -eq 0]; then

```
    echo "Found!"
```

elif [$? -eq 1]; then

```
    echo "Not found!"
```

fi

Shorter: use directly in if

if grep -q "pattern" file.txt; then

```
    echo "Found!"
```

```
fi

# Use with && and ||

grep -q "boss" enemies.txt && echo "Boss exists!" || echo "No boss"
```

Exercise: grep Practice

```
1. Create a file with 10 lines of text and search for a specific word.
2. Use grep -i for case-insensitive search.
3. Count how many lines contain a pattern using grep -c .
4. Use grep -E to search for multiple patterns with | .
5. Find all .sh files containing the word "function".
6. Use grep -q in an if statement to check if a word exists.
```

Common Mistake: Not Using -q in Conditions

```
When using grep in an if statement, always use -q (quiet mode). Without it, grep prints matching
lines to stdout, cluttering your output:



# Messy — prints all matching lines
if grep "pattern" file.txt; then
echo "Found"
fi

# Clean — no output, just exit code
if grep -q "pattern" file.txt; then
echo "Found"
fi
```

Project: Word Game Validator

Build a word game that validates guesses using grep:

```
#!/bin/bash
# wordgame.sh — grep-powered word game

# Create a word list if it doesn't exist
if [[ ! -f words.txt ]]; then
cat > words.txt << 'EOF'
dragon
wizard
castle
sword
potion
shield
goblin
knight
sorcery
potion
kingdom
treasure
dungeon
monster
spells
potion
EOF
fi

# ── Game Functions ──────────────────────────────────

# Check if word is in the dictionary
is_valid_word() {
grep -qx "$1" words.txt
}

# Count occurrences of a letter in the target word
count_letter() {
local word="$1"
local letter="$2"
echo "$word" | grep -o "$letter" | wc -l
}




# ── Main Game ──────────────────────────────────────
echo "═══ Word Wizard ═══"
echo "Guess words to score points!"
```

```
echo "Type 'quit' to exit."
echo ""

score=0
round=1

while true; do
# Pick a random word from the list
word=$(shuf -n 1 words.txt)
word_len=${#word}

echo "── Round $round ──"
echo "Find a word with $word_len letters."
echo "Hint: it contains the letter '$(echo $word | grep -o . | shuf -n 1)'"

read -p "Your guess: " guess

# Check for quit
[[ "$guess" == "quit" ]] && break

# Validate length
if (( ${#guess} != word_len )); then
echo "Wrong length! Need $word_len letters."
continue
fi

# Check if it's a valid word
if ! is_valid_word "$guess"; then
echo "'$guess' is not in the dictionary!"
continue
fi

# Check if correct
if [[ "$guess" == "$word" ]]; then
points=$((word_len * 10))
score=$((score + points))
echo "Correct! +$points points (Total: $score)"
else
echo "Wrong! The word was: $word"
fi

echo ""
((round++))
done
```

```
echo ""
echo "═══ Game Over ═══"
echo "Final score: $score"
echo "Rounds played: $((round - 1))"
```

This word game uses grep to validate words against a dictionary file, shuf to pick random words,

and grep -o to extract letters — showing how grep is a core tool for text-based game logic.

sed — Stream Editor

What You Will Learn

sed (Stream Editor) transforms text. It can substitute, delete, insert, and modify text on a line-by-

line basis. In game development, sed is perfect for modifying save files, processing config files, and

transforming game data.

Basic Substitution

```
# Replace first occurrence on each line
sed 's/old/new/' file.txt

# Replace all occurrences on each line
sed 's/old/new/g' file.txt

# Case-insensitive replacement
sed 's/old/new/gi' file.txt

# Replace only the 2nd occurrence per line
sed 's/old/new/2' file.txt

# Replace and print only changed lines
sed -n 's/old/new/p' file.txt

# In-place editing (modifies the file directly)
sed -i 's/old/new/g' file.txt

# In-place with backup
sed -i.bak 's/old/new/g' file.txt



sed substitution anatomy:

sed 's/old/new/g' file.txt
| | |    | |
| | |    | └── g = global (all occurrences per line)
| | |    └─────── replacement text
| | └──────────── pattern to find
| └────────────── s = substitute command
└──────────────── quoted expression

Delimiter can be changed:
sed 's|/usr/local|/opt|g'     (useful for paths)
sed 's#old#new#g'
```

Addressing: Which Lines to Edit

```
# Operate on a specific line number
sed '3s/old/new/' file.txt

# Operate on a range of lines
sed '5,10s/old/new/g' file.txt

# Operate on the last line
sed '$s/old/new/' file.txt

# Operate on lines matching a pattern
sed '/error/s/old/new/' file.txt

# Operate on lines NOT matching a pattern
sed '/error/!s/old/new/' file.txt

# From a pattern to end of file
sed '/start/,$s/old/new/g' file.txt
```

Other sed Commands

```
# Delete lines
sed '5d' file.txt                 # delete line 5
sed '/pattern/d' file.txt         # delete matching lines
sed '/^$/d' file.txt              # delete empty lines
sed '1,5d' file.txt              # delete lines 1-5

# Print specific lines
sed -n '5,10p' file.txt          # print lines 5-10
sed -n '/pattern/p' file.txt      # print matching lines only

# Insert text before a line
sed '3i\Inserted text' file.txt
```

Append text after a line

sed '3a\Appended text' file.txt

Change (replace) an entire line

sed '3c\Replaced line' file.txt

Print line numbers

sed '=' file.txt

Quit after N lines

sed '5q' file.txt	**# process only first 5 lines**

Multiple sed Commands

Multiple commands with -e

sed -e 's/cat/dog/g' -e 's/bird/fish/g' file.txt

Or use a script file

cat > script.sed << 'EOF'

s/cat/dog/g

s/bird/fish/g

/^$/d

EOF

sed -f script.sed file.txt

Or use semicolons (no -e needed)

sed 's/cat/dog/g; s/bird/fish/g' file.txt

Group with {}

sed '/error/{ s/old/new/g; /debug/d; }' file.txt

Capture Groups in sed

Use \(\) in basic regex, ( ) in extended regex

Refer to groups with \1, \2, etc.

Swap two words

echo "hello world" | sed -E 's/(.*) (.*)/\2 \1/'	**# world hello**

Reformat date

echo "2024-01-15" | sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\2/\3/\1/'	**# 01/15/2024**

Add prefix to every line

sed 's/^/>> /' file.txt

Add suffix

sed 's/$/ <<</' file.txt

Wrap field in quotes (CSV processing)

echo "Alice,30,Warrior" | sed -E 's/([^,]+)/"\1"/'	**# "Alice",30,Warrior**

Practical sed Examples for Games

Update a value in a config file

sed -i 's/^player_hp=.*/player_hp=100/' game.cfg

Remove all comment lines (starting with #)

sed '/^#/d' config.txt

Remove empty lines

sed '/^$/d' file.txt

Replace tab with spaces

sed 's/\t/	**/g' file.txt**

Convert to lowercase

sed 's/.*/\L&/' file.txt	# GNU sed extension

Number all lines

sed '=' file.txt | sed 'N;s/\n/\t/'

Extract a section between markers

sed -n '/START/,/END/p' file.txt

Delete lines between markers

sed '/START/,/END/d' file.txt

Duplicate every line

sed 'p' file.txt | sed 'N'

Tip: sed -i Creates No Backup by Default

```
sed -i modifies files in place with no backup. If your pattern is wrong, data is lost. Always test
without -i first, or use sed -i.bak to create a backup:


# Test first (output to screen)
sed 's/old/new/g' important.cfg

# Apply with backup
sed -i.bak 's/old/new/g' important.cfg
# Creates important.cfg.bak as backup
```

Exercise: sed Practice

```
1. Use sed to replace all "cat" with "dog" in a file.
2. Delete all empty lines from a file.
3. Use sed -n to print lines 5 through 10.
4. Insert a line of text before line 3.



5. Use capture groups to swap first and last names in "First Last" format.
6. Use sed -i with a backup to modify a config file.
```

Common Mistake: Wrong Delimiter in Paths

When substituting paths that contain / , the / delimiter conflicts. Use a different delimiter:

```
# Confusing — too many slashes
sed 's-/usr/local-/opt/g' file.txt     # Hard to read!

# Clean — use | as delimiter
sed 's|/usr/local|/opt|g' file.txt

# Also works
sed 's#/usr/local#/opt#g' file.txt
sed 's@/usr/local@/opt@g' file.txt
```

Project: Save File Editor

Build a save file editor using sed:

```
#!/bin/bash
# save_editor.sh — edit game save files with sed

# Create a save file
cat > savegame.dat << 'EOF'
# Game Save File
player_name=Hero
player_hp=85
player_gold=340
player_level=7
player_xp=1250
current_room=dungeon_3
inventory=sword,potion,key
EOF

echo "═══ Save File Editor ═══"
echo "Original save:"
cat savegame.dat
echo ""

# Update player HP
sed -i 's/^player_hp=.*/player_hp=100/' savegame.dat
echo "HP restored to 100!"

# Add gold
current_gold=$(grep '^player_gold=' savegame.dat | cut -d= -f2)
new_gold=$((current_gold + 500))
sed -i "s/^player_gold=.*/player_gold=$new_gold/" savegame.dat



echo "Gold increased to $new_gold!"

# Add item to inventory
sed -i 's/^inventory=\(.*\)/inventory=\1,shield/' savegame.dat
echo "Added shield to inventory!"

# Add a timestamp line
sed -i '/^# Game Save File/a last_saved='"$(date)" savegame.dat
echo "Added save timestamp!"

# Remove XP line (player reset)
# sed -i '/^player_xp/d' savegame.dat

echo ""
```

```
echo "Modified save:"
cat savegame.dat

# Clean up
rm -f savegame.dat.bak
```

This save editor demonstrates real-world sed usage: updating values in config-like files, appending to comma-separated lists, inserting new lines, and using capture groups — all operations you will need when managing game state files.

awk — Text Processing

What You Will Learn

awk is a powerful text processing tool that treats each line as a record and each word as a field. It is

perfect for processing tabular data, calculating statistics, building score tables, and implementing

game data queries.

Basic awk Usage

```
# Print every line (like cat)
awk '{print}' file.txt

# Print specific field
awk '{print $1}' file.txt        # first field
awk '{print $2, $3}' file.txt   # fields 2 and 3

# Print last field
awk '{print $NF}' file.txt      # NF = number of fields

# Print line number and line
awk '{print NR, $0}' file.txt   # NR = line number

# Print total number of fields per line
awk '{print NF}' file.txt




awk field model:

Input line:   Alice 95 Warrior 12

$1    $2    $3      $4
Alice 95   Warrior 12

$0 = entire line: "Alice 95 Warrior 12"
NF = 4 (number of fields)
NR = current line number

Default field separator: whitespace (spaces/tabs)
Can be changed with -F flag
```

Field Separators

```
# Use comma as separator (CSV)
awk -F',' '{print $1, $3}' data.csv

# Use colon (like /etc/passwd)
awk -F':' '{print $1}' /etc/passwd

# Use tab
awk -F'\t' '{print $1}' data.tsv

# Multiple separators
awk -F'[,;]' '{print $1}' data.txt

# Set separator inside awk
awk 'BEGIN {FS=","} {print $1}' data.csv

# Set output separator
awk 'BEGIN {FS=","; OFS=" | "} {print $1, $2, $3}' data.csv
```

Patterns and Actions

awk programs have the structure: pattern -{ action -} . If pattern matches, the action runs.

```
# Print lines where field 2 is greater than 50
awk '$2 > 50 {print}' scores.txt

# Print lines matching a regex
awk '/error/ {print}' log.txt

# Print specific fields from matching lines
awk '/Warrior/ {print $1, $3}' players.txt

# Print lines where field 1 matches pattern
awk '$1 ~ /^A/ {print}' names.txt   # names starting with A

# Print lines where field 1 does NOT match
awk '$1 !~ /^A/ {print}' names.txt

# Multiple patterns
awk '/error/ {print "ERR:", $0} /warning/ {print "WARN:", $0}' log.txt
```

BEGIN and END Blocks

```
# BEGIN runs before processing any lines
# END runs after all lines are processed

awk 'BEGIN {print "Start"} {print} END {print "Done"}' file.txt




# Calculate sum of column 2
awk '{sum += $2} END {print "Total:", sum}' scores.txt

# Calculate average
awk '{sum += $2; count++} END {print "Average:", sum/count}' scores.txt

# Find maximum value
awk 'NR > 1 {max = $2} $2 > max {max = $2} END {print "Max:", max}' scores.txt

# Count lines
awk 'END {print NR " lines"}' file.txt

# Multiple calculations
awk 'BEGIN {
print "═══ Score Analysis ═══"
print "Name\tScore\tGrade"
}
{
sum += $2
if ($2 >= 90) grade = "A"
else if ($2 >= 80) grade = "B"
else if ($2 >= 70) grade = "C"
else grade = "F"
print $1 "\t" $2 "\t" grade
}
END {
print "═══════════════════════"
print "Average:", sum/NR
}' students.txt
```

awk Control Structures

```
# if-else
awk '{if ($2 > 80) print $1, "Pass"; else print $1, "Fail"}' scores.txt

# for loop
awk '{for (i=1; i<=NF; i++) print $i}' data.txt

# while loop
awk '{i=1; while (i<=NF) {print $i; i++}' data.txt

# Arrays in awk
awk '{names[NR] = $1} END {for (i=1; i<=NR; i++) print names[i]}' file.txt

# Associative arrays (count by category)
awk '{count[$1]-+} END {for (key in count) print key, count[key]}' data.txt
```

Built-in awk Variables

Variable	Description
$0	Entire current line

$1 , $2 , ...	Individual fields
NF	Number of fields in current line
NR	Current line (record) number

FS	Input field separator
OFS	Output field separator
RS	Input record separator (default: newline)
ORS	Output record separator
FILENAME	Current input file name
FNR	Line number within current file

awk One-Liners for Games

```
# Sort scores by value (field 2) descending
awk '{print $2, $1}' scores.txt | sort -rn | awk '{print $2, $1}'

# Top 3 scores
awk '{print $2, $1}' scores.txt | sort -rn | head -3 | awk '{print $2, $1}'

# Find players above a threshold
awk '$2 > 1000 {print $1, $2}' scores.txt

# Sum all gold values
awk -F',' '{gold[$1] += $3} END {for (p in gold) print p, gold[p]}' players.csv

# Count players per class
awk '{class[$3]-+} END {for (c in class) print c, class[c]}' players.txt

# Format as a table
awk 'BEGIN {printf "%-15s %5s %5s\n", "Name", "Score", "Level"}
{printf "%-15s %5d %5d\n", $1, $2, $3}' players.txt
```

Tip: awk printf for Formatting

```
awk has printf for formatted output — essential for aligning game tables and score boards:


awk 'BEGIN {
printf "%-20s %8s %8s\n", "Player", "Score", "Level"
printf "%s\n", "----------------------------------------"
}
{




printf "%-20s %8d %8d\n", $1, $2, $3
}' players.txt
```

Exercise: awk Practice

```
1. Create a CSV file with player data: name, score, level.
2. Use awk to print only the names.
3. Calculate the total score across all players.
4. Find the player with the highest score using awk .
5. Use awk with printf to format a score table.
6. Count how many players are at each level using associative arrays.
```

Common Mistake: Forgetting BEGIN for Variables

If you need to initialize variables before processing, use BEGIN. Otherwise, they initialize to 0/empty on first use:

```
# Works — sum starts at 0 automatically
awk '{sum += $2} END {print sum}' file.txt

# Need BEGIN for other setup
awk 'BEGIN {FS=","; max=0} $2 > max {max = $2} END {print max}' file.txt
```

Project: Leaderboard System

Build a complete leaderboard system with awk:

```
#!/bin/bash
# leaderboard.sh — awk-powered leaderboard

# Create a scores database
cat > scores.db << 'EOF'
Alice,12500,Warrior,15
Bob,8200,Mage,12
Carol,18900,Rogue,18
Dave,5500,Warrior,8
Eve,14300,Mage,14
Frank,9800,Rogue,11
Grace,21000,Warrior,20
Heidi,7200,Mage,9
EOF
```

echo "═══ Leaderboard System ═══"

echo ""

1. Top 5 scores

echo "── Top 5 Players ──"

awk -F',' 'BEGIN {

```
printf "%-15s %8s %10s %5s\n", "Player", "Score", "Class", "Level"
print "-------------------------------------------------"
```

}

{

```
scores[NR] = $2
data[NR] = $0
```

}

END {

```
# Simple bubble sort by score
for (i = 1; i <= NR; i++) {
for (j = i + 1; j <= NR; j++) {
if (scores[j] > scores[i]) {
tmp = scores[i]; scores[i] = scores[j]; scores[j] = tmp
tmp = data[i]; data[i] = data[j]; data[j] = tmp
}
}
}
for (i = 1; i <= 5 && i <= NR; i++) {
split(data[i], f, ",")
printf "%-15s %8d %10s %5d\n", f[1], f[2], f[3], f[4]
}
```

}' scores.db

echo ""

2. Average score

echo "── Statistics ──"

awk -F',' '{

```
total += $2
count++
if ($2 > max) { max = $2; max_name = $1 }
if (min == 0 || $2 < min) { min = $2; min_name = $1 }
```

}

END {

```
printf "Total players: %d\n", count
printf "Average score: %.1f\n", total/count
printf "Highest: %s (%d)\n", max_name, max
printf "Lowest: %s (%d)\n", min_name, min
```

}' scores.db

echo ""

3. Players by class

echo "── Players by Class ──"

awk -F',' '{

```
class_count[$3]-+
class_total[$3] += $2
```

}

END {

```
printf "%-12s %8s %12s\n", "Class", "Players", "Avg Score"
print "---------------------------------------"
for (c in class_count) {




printf "%-12s %8d %12.1f\n", c, class_count[c], class_total[c]/class_count[c]
}
}' scores.db

echo ""

# 4. Search for a player
echo "── Search ──"
read -p "Enter player name: " search
awk -F',' -v name="$search" '
tolower($1) ~ tolower(name) {
printf "Found: %s — Score: %d, Class: %s, Level: %d\n", $1, $2, $3, $4
found = 1
}
END {
if (!found) print "Player not found."
}
' scores.db

rm -f scores.db
```

This leaderboard uses awk for sorting, statistics, grouping, searching, and formatted output — demonstrating awk's full power as a data processing tool for game systems.

Interactive Bash

What You Will Learn

How to create rich, interactive terminal experiences: colored text, cursor control, screen clearing,

menus, single-key input, and raw terminal mode. These are the tools that make terminal games feel

like games, not just text-processing scripts.

ANSI Color Codes

Terminal colors are controlled by ANSI escape sequences embedded in text:

```
# ANSI escape sequence format:
# \033<<code>m

# Foreground colors
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
PURPLE='\033[0;35m'
CYAN='\033[0;36m'
WHITE='\033[1;37m'
NC='\033[0m'     # No Color (reset)

# Usage
echo -e "${RED}Error: ${NC}Something went wrong"
echo -e "${GREEN}Success!${NC}"
echo -e "${YELLOW}Warning: ${NC}Low health!"
echo -e "${CYAN}Info: ${NC}Loading game..."

# Background colors
BG_RED='\033[41m'
BG_GREEN='\033[42m'
BG_BLUE='\033[44m'
echo -e "${BG_RED} GAME OVER ${NC}"
```

Code	Color	Bold Code
30	Black	1;30
31	Red	1;31
32	Green	1;32
33	Yellow	1;33

34	Blue	1;34
35	Magenta	1;35
36	Cyan	1;36
37	White	1;37

0	Reset/Normal	—

Text Styles

```
# Style codes
echo -e '\033[1mBold\033[0m'             # Bold
echo -e '\033[4mUnderline\033[0m'        # Underline
echo -e '\033[7mReverse\033[0m'          # Reverse video (swap fg/bg)
echo -e '\033[2mDim\033[0m'              # Dim/faint
echo -e '\033[5mBlink\033[0m'            # Blink (may not work everywhere)

# 256-color (on modern terminals)
for i in $(seq 0 255); do
printf "\033[38;5;${i}m%3d " "$i"
(( (i+1) % 16 == 0 )) && echo ""
done
printf '\033[0m\n'
```

Using tput for Portability

tput provides terminal-independent control:

```
# Colors (using tput)
RED=$(tput setaf 1)
GREEN=$(tput setaf 2)
YELLOW=$(tput setaf 3)
BLUE=$(tput setaf 4)
CYAN=$(tput setaf 6)
BOLD=$(tput bold)
UNDERLINE=$(tput smul)
RESET=$(tput sgr0)

echo "${RED}${BOLD}Game Over!${RESET}"
echo "${GREEN}You win!${RESET}"
echo "${BLUE}${UNDERLINE}Instructions${RESET}"

# Cursor movement
tput cup 10 5        # Move cursor to row 10, col 5
tput cup 0 0         # Move to top-left

# Clear screen
tput clear           # or: clear




# Get terminal size
rows=$(tput lines)
cols=$(tput cols)
echo "Terminal: $cols x $rows"

# Hide/show cursor
tput civis            # Hide cursor
tput cnorm            # Show cursor

# Save/restore cursor position
tput sc              # Save position
tput rc              # Restore position
```

Tip: Always Reset Colors

```
Always end colored output with a reset code ( \033[0m or tput sgr0 ). If you forget, the color
"leaks" into subsequent output, making everything after it the same color.
```

Screen Control

```
# Clear the screen
clear                     # Simple clear
printf '\033[2J'         # ANSI clear screen
printf '\033[2J\033[H'   # Clear and move cursor to home

# Move cursor
printf '\033[H'          # Home (top-left)
printf '\033[5;10H'      # Row 5, column 10
printf '\033[2A'         # Up 2 lines
printf '\033[3B'         # Down 3 lines
printf '\033[5C'         # Right 5 columns
printf '\033[5D'         # Left 5 columns

# Clear portions
printf '\033[2K'         # Clear entire line
printf '\033[K'          # Clear from cursor to end of line
printf '\033[1K'         # Clear from start to cursor
```

Single-Key Input

Read a single character without waiting for Enter:

```
# Read single keypress
read -n 1 -s key
echo "You pressed: $key"

# With a prompt
read -n 1 -s -p "Press any key..." key



echo ""
echo "Key: $key"

# With timeout
read -t 5 -n 1 -s key
if [[ $? -eq 0 ]]; then
echo "You pressed: $key"
else
echo "Timeout!"
fi

# Game-style key reading
read -n 1 -s -p "Move (WASD): " move
echo ""
case "$move" in
[Ww]) echo "Moving up" ;;
[Ss]) echo "Moving down" ;;
[Aa]) echo "Moving left" ;;
[Dd]) echo "Moving right" ;;
[Qq]) echo "Quitting" ;;
*)   echo "Unknown key" ;;
esac
```

Raw Terminal Mode (No Echo)

For real-time games, you may want to disable echo and line buffering:

```
# Disable echo and canonical mode (line buffering)
stty -echo -icanon

# Read a single key instantly (no Enter, no display)
key=$(dd bs=1 count=1 2>/dev/null)

# ALWAYS restore terminal settings when done!
stty sane
# or:
stty echo icanon

# Safe pattern with trap
cleanup() {
stty sane
clear
}
trap cleanup EXIT

stty -echo -icanon
# ... game loop ...
# Terminal is restored on exit via trap



Terminal modes:

Normal (cooked) mode:
- Line buffered (waits for Enter)
- Echo on (characters displayed)
- Signal processing on (Ctrl+C works)
Good for: menus, text input, quizzes

Raw mode:
- No buffering (instant keypress)
- No echo (characters not displayed)
- No signal processing (Ctrl+C may not work!)
Good for: arcade games, real-time movement

Always restore with: stty sane
Use trap to ensure cleanup on exit!
```

Menus with select

PS3="Choose: "

select option in "New Game" "Load Game" "Settings" "Quit"; do

```
case "$option" in
"New Game")
echo "Starting new game..."
break
;;
"Load Game")
echo "Loading..."
break
;;
"Settings")
echo "Settings menu"
;;
"Quit")
echo "Goodbye!"
exit 0
;;
*)
echo "Invalid choice. Please select 1-4."
;;
esac
```

done

Box Drawing

Unicode box drawing characters

draw_box() {

```
local msg="$1"
local width=$((${#msg} + 4))



local line=""
for ((i=0; i<width; i++)); do line+="═"; done

echo "╔$line╗"
echo "║ $msg ║"
echo "╚$line╝"
```

}

draw_box "WELCOME TO THE GAME"

draw_box "Level 5: Dark Dungeon"

Progress bar

progress_bar() {

```
local current=$1
local max=$2
local width=20
local filled=$((current * width / max))
printf "["
for ((i=0; i<filled; i++)); do printf "█"; done
for ((i=filled; i<width; i++)); do printf "░"; done
printf "] %d/%d\n" "$current" "$max"
```

}

progress_bar 75 100

progress_bar 3 10

Exercise: Interactive Bash

```
1. Print colored text in at least 4 different colors.
2. Read a single keypress and respond to it.
3. Create a menu using select .
4. Draw a progress bar that fills based on a percentage.
5. Use tput to center text on the screen.
6. Build a raw-mode input handler with proper cleanup on exit.
```

Common Mistake: Forgetting to Restore Terminal Settings

If you use stty -echo -icanon and your script crashes or exits without restoring, the terminal will be left in a broken state — no echo, no line buffering. Always use a trap:

```
# ALWAYS do this when modifying terminal settings
trap 'stty sane' EXIT
stty -echo -icanon
# ... game code ...
# Terminal auto-restored on exit, even on crash/Ctrl+C
```

Project: Animated Splash Screen

Create an animated game splash screen with colors and effects:

```
#!/bin/bash
# splash.sh — animated splash screen

# Color setup
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
CYAN='\033[0;36m'
BOLD='\033[1m'
BLINK='\033[5m'
NC='\033[0m'

# Cleanup on exit
cleanup() {
stty sane
tput cnorm # Show cursor
clear
}
trap cleanup EXIT

# Hide cursor
tput civis

# Clear screen
clear

# Get terminal size
rows=$(tput lines)
cols=$(tput cols)
center_row=$((rows / 2 - 3))
center_col=$((cols / 2 - 15))

# Draw the title
tput cup $center_row $center_col
echo -e "${CYAN}${BOLD}╔═══════════════════════════════════╗"
tput cup $((center_row + 1)) $center_col
echo -e "${CYAN}${BOLD}║                                   ║"
tput cup $((center_row + 2)) $center_col
echo -e "${YELLOW}${BOLD}║     TERMINAL QUEST v1.0       ║"
tput cup $((center_row + 3)) $center_col
echo -e "${CYAN}${BOLD}║                                   ║"
tput cup $((center_row + 4)) $center_col
echo -e "${CYAN}${BOLD}╚═══════════════════════════════════╝"

# Animated loading bar
```

```
bar_row=$((center_row + 7))
bar_col=$((center_col + 2))
bar_width=28
tput cup $bar_row $bar_col
echo -ne "${GREEN}Loading: ["
for ((i=0; i<bar_width; i++)); do
printf "░"
done



echo -ne "]${NC}"

# Animate the bar
for ((i=0; i<bar_width; i++)); do
tput cup $bar_row $((bar_col + 10 + i))
echo -ne "${GREEN}█${NC}"
sleep 0.05
done

# Flash "Press Start"
for flash in 1 2 3; do
tput cup $((bar_row + 3)) $((center_col + 8))
echo -ne "${BOLD}Press Enter to Start${NC}"
sleep 0.5
tput cup $((bar_row + 3)) $((center_col + 8))
echo -ne "                      "
sleep 0.3
done

tput cup $((bar_row + 3)) $((center_col + 8))
echo -ne "${BLINK}${BOLD}Press Enter to Start${NC}"

read -s
clear
echo -e "${GREEN}Welcome to Terminal Quest!${NC}"
```

This splash screen demonstrates colors, cursor positioning, animation, box drawing, blinking text,

and cursor hiding — the visual toolkit for making terminal games feel polished and professional.

Chapter 20

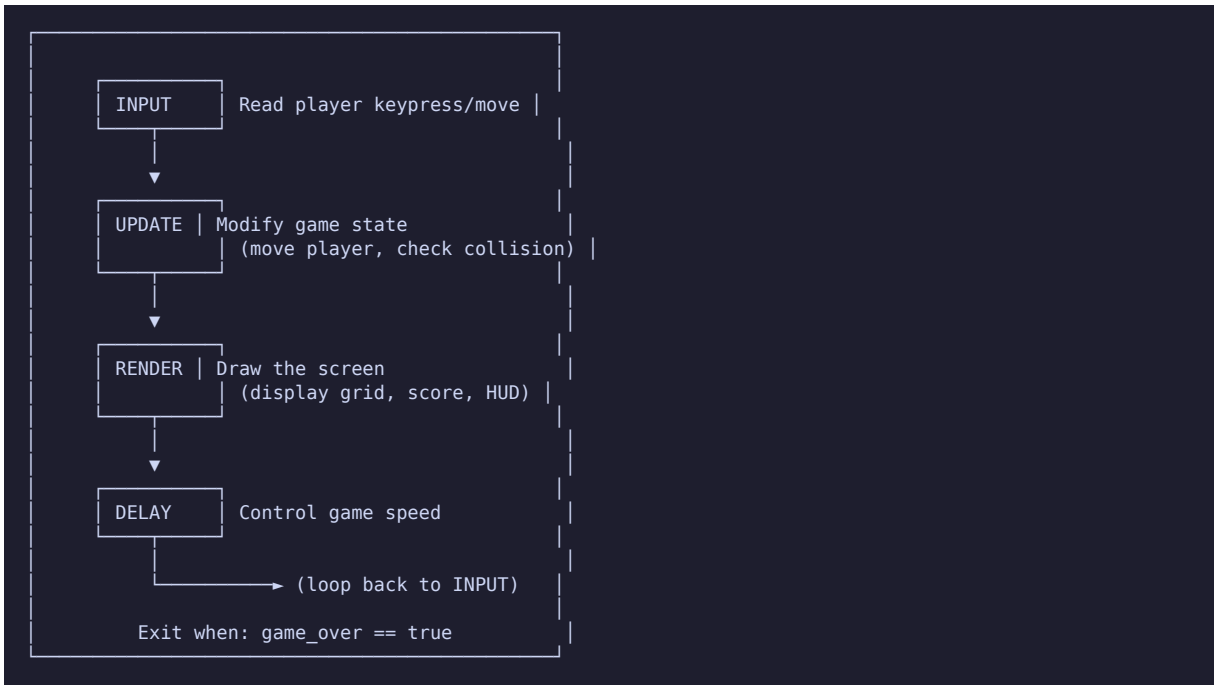
Game Loops and Architecture

What You Will Learn

Every game has a game loop — the core cycle of reading input, updating state, and rendering output. You will learn game loop patterns, state management, timing, and how to structure a terminal game for maintainability.

The Game Loop

The game loop is the heartbeat of every game. It repeats continuously until the game ends:



Basic Game Loop in Bash

#!/bin/bash

game_loop.sh — basic game loop pattern

game_over=false

score=0

while [["$game_over" == "false"]]; do

```
# 1. INPUT — read player input
read -n 1 -s -t 1 key    # -t 1 = 1 second timeout

# 2. UPDATE — modify game state based on input
case "$key" in
q) game_over=true ;;
w) echo "Moving up" ;;
s) echo "Moving down" ;;
a) echo "Moving left" ;;
d) echo "Moving right" ;;
"") ;; # timeout, no input
esac

# Game logic (move enemies, check collisions, etc.)
# ...

# 3. RENDER — draw the current state
# clear
# echo "Score: $score"
# ... draw the game screen ...

# 4. DELAY — control frame rate
sleep 0.1
```

done

echo "Final score: $score"

echo "Thanks for playing!"

Turn-Based vs Real-Time Loops

```
Turn-Based (waits for input):
while not game_over:
read input (blocks until player acts)
update state
render
→ Player controls the pace. Good for: RPGs, strategy, puzzles.

Real-Time (continuous loop with timeout):
while not game_over:
read input (timeout after N ms)
update state (always runs, even without input)
render
sleep (frame delay)
→ Game runs at fixed speed. Good for: arcade, action, snake.
```

Turn-Based Game Loop

#!/bin/bash

turn_based.sh — turn-based game loop

player_x=1

player_y=1

game_over=false

while [["$game_over" == "false"]]; do

```
# RENDER
clear
echo "═══ Dungeon Crawl ═══"
echo "Position: ($player_x, $player_y)"
echo "Controls: WASD to move, Q to quit"
echo ""

# Draw a 5x5 grid with player
for ((y=0; y<5; y++)); do
row=""
for ((x=0; x<5; x++)); do
if [[ $x -eq $player_x && $y -eq $player_y ]]; then
row+=" @ "
else
row+=" . "
fi
done
echo "$row"
done

# INPUT (blocks until player presses a key)
read -n 1 -s key
echo ""

# UPDATE




case "$key" in
[Ww]) (( player_y > 0 )) && ((player_y--)) ;;
[Ss]) (( player_y < 4 )) && ((player_y++)) ;;
[Aa]) (( player_x > 0 )) && ((player_x--)) ;;
[Dd]) (( player_x < 4 )) && ((player_x++)) ;;
[Qq]) game_over=true ;;
esac
```

done

echo "Game Over!"

Real-Time Game Loop

#!/bin/bash

realtime.sh — real-time game loop with timed input

Setup

trap 'stty sane; clear; echo "Game Over"' EXIT

stty -echo -icanon

player_x=5

score=0

game_over=false

Main loop

while [["$game_over" == "false"]]; do

```
# RENDER
printf '\033[H\033[2J' # clear and home
echo "═══ Dodge Game ═══ Score: $score"
echo ""

# Draw a line of characters with player position
for ((x=0; x<20; x++)); do
if [[ $x -eq $player_x ]]; then
printf "\033[32m@\033[0m"
else
printf "-"
fi
done
echo ""
echo "A/D to move, Q to quit"

# INPUT (non-blocking, with timeout)
read -t 0.1 -n 1 key 2>/dev/null

# UPDATE
case "$key" in
[Aa]) (( player_x > 0 )) && ((player_x--)) ;;
[Dd]) (( player_x < 19 )) && ((player_x++)) ;;
[Qq]) game_over=true ;;
esac

((score++))

# Frame delay




sleep 0.05
done
```

State Management

Game state is all the data that describes the current game situation. Organize it clearly:

```
#!/bin/bash
# state.sh — organized game state

# ── Player State ──────────────────────────────────
player_name="Hero"
player_x=0
player_y=0
player_hp=100
player_max_hp=100
player_gold=0
player_level=1
player_xp=0

# ── World State ───────────────────────────────────
current_room=1
turn_count=0
difficulty="normal"
MAP_WIDTH=10
MAP_HEIGHT=10

# ── Game State ────────────────────────────────────
game_over=false
game_won=false
game_paused=false

# ── Arrays ────────────────────────────────────────
declare -A inventory
declare -A world_map
declare -a enemies

# ── Functions to modify state ─────────────────────
move_player() {
local dx=$1 dy=$2
local new_x=$((player_x + dx))
local new_y=$((player_y + dy))

# Check boundaries
if (( new_x >= 0 && new_x < MAP_WIDTH && new_y >= 0 && new_y < MAP_HEIGHT )); then
player_x=$new_x
player_y=$new_y
return 0
else
echo "You can't go that way."
return 1
fi
```

```
}

take_damage() {
local amount=$1
player_hp=$((player_hp - amount))




if (( player_hp == 0 )); then
player_hp=0
game_over=true
echo "You have died!"
fi
```

}

add_gold() {

```
local amount=$1
player_gold=$((player_gold + amount))
echo "Found $amount gold!"
```

}

Game Speed and Timing

sleep with fractional seconds (GNU coreutils)

sleep 0.1	# 100ms
sleep 0.05	# 50ms
sleep 1.5	# 1.5 seconds

Calculate frame rate

60 FPS = ~16.67ms per frame

30 FPS = ~33.33ms per frame

10 FPS = 100ms per frame (typical for terminal games)

For a 10 FPS game:

FRAME_DELAY=0.1

Animation timing

for ((frame=0; frame<10; frame++)); do

```
echo -ne "\rFrame $frame"
sleep $FRAME_DELAY
```

done

echo ""

Countdown timer

countdown() {

```
local seconds=$1
while (( seconds > 0 )); do
printf "\rTime remaining: %2d" "$seconds"
sleep 1
((seconds--))
done
echo "\rTime's up!        "
```

}

countdown 5

Save and Load System

Save game state to a file

save_game() {

```
local savefile="save_$(date +%s).dat"
cat > "$savefile" << EOF
```

Save file created: $(date)

player_name=$player_name

player_hp=$player_hp

player_gold=$player_gold

player_level=$player_level

player_xp=$player_xp

current_room=$current_room

game_score=$score

EOF

```
echo "Game saved to $savefile"
```

}

Load game state from a file

load_game() {

```
local savefile="$1"
if [[ ! -f "$savefile" ]]; then
echo "Save file not found!"
return 1
fi
# Source the file to load variables
source "$savefile"
echo "Game loaded: $player_name (Level $player_level)"
```

}

Alternative: use a simple key=value format with source

This is the easiest approach in Bash

```
Game architecture overview:

┌──────────────────────────────────────────┐
│            Main Game Loop                │
│  ┌─────┐  ┌────────┐  ┌────────┐         │
│  │Input│─▶│Update  │─▶│Render  │         │
│  └─────┘  └────────┘  └────────┘         │
│                │                         │
│     ┌──────────┼──────────┐              │
│     ▼          ▼          ▼              │
│  ┌──────┐  ┌──────┐  ┌──────────┐        │
│  │Player│  │ World│  │ Enemy    │        │
│  │ State│  │ State│  │ System   │        │
│  └──────┘  └──────┘  └──────────┘        │
│     │          │          │              │
│     ▼          ▼          ▼              │
│  ┌──────────────────────────┐            │
│  │       Save/Load System   │            │
│  └──────────────────────────┘            │
└──────────────────────────────────────────┘

Each system is a set of functions that read/modify
the game state variables. The loop coordinates them.
```

Exercise: Game Loops

```
1. Write a turn-based loop that moves a player on a 5x5 grid.
2. Modify it to place a goal tile and detect when the player reaches it.
3. Add an enemy that moves randomly each turn.
4. Implement a save/load system using a file.
5. Create a countdown timer that limits the number of turns.
6. Convert the turn-based loop to a real-time loop with timed input.
```

Common Mistake: Blocking Input in Real-Time Games

In real-time games, read without a timeout blocks the entire loop. Always use read -t with a short timeout:

```
# BAD — blocks the loop until player presses a key
read -n 1 -s key

# GOOD — times out after 0.1s, allowing the loop to continue
read -t 0.1 -n 1 -s key 2>/dev/null
```

Project: Mini Snake Game

Build a simplified Snake game using a real-time loop:

```
#!/bin/bash
# snake.sh — simplified terminal snake game

# Cleanup
cleanup() {
stty sane
tput cnorm
clear
echo "═══ Snake Game ═══"
echo "Final Score: $score"
echo "Thanks for playing!"
}
trap cleanup EXIT

# Setup terminal
stty -echo -icanon
tput civis

# Game settings
WIDTH=40
```

HEIGHT=15

SNAKE_CHAR="█"

FOOD_CHAR="*"

SPEED=0.15

Initial state

snake_x=20

snake_y=7

dir_x=1

dir_y=0

```
food_x=10
food_y=5
score=0
game_over=false
# Snake body (array of "x,y" positions)
body=()
place_food() {
    food_x=$((RANDOM % (WIDTH - 2) + 1))
    food_y=$((RANDOM % (HEIGHT - 2) + 1))
}
# Main game loop
while [[ "$game_over" == "false" ]]; do
    # RENDER
    printf '\033[H\033[2J'
    echo "Snake — Score: $score"

    # Draw top border
    echo "+$(printf '=%.0s' $(seq 1 $WIDTH))+"

    for ((y=0; y<HEIGHT; y++)); do
    printf "|"
    for ((x=0; x<WIDTH; x++)); do
    if [[ $x -eq $snake_x && $y -eq $snake_y ]]; then
    printf "\033[32m%s\033[0m" "$SNAKE_CHAR"
    elif [[ $x -eq $food_x && $y -eq $food_y ]]; then
    printf "\033[31m%s\033[0m" "$FOOD_CHAR"
    else
    is_body=false
    for pos in "${body[@]}"; do
    bx="${pos%,*}"
    by="${pos#*,}"
    if [[ $x -eq $bx && $y -eq $by ]]; then
    printf "\033[32m%s\033[0m" "$SNAKE_CHAR"
    is_body=true
    break
    fi
    done
    [[ "$is_body" == "false" ]] && printf " "
    fi
    done
    echo "|"
    done

    # Draw bottom border
    echo "+$(printf '=%.0s' $(seq 1 $WIDTH))+"
    echo "WASD to move, Q to quit"




    # INPUT
    read -t "$SPEED" -n 1 key 2>/dev/null
    case "$key" in
    [Ww]) dir_x=0; dir_y=-1 ;;
    [Ss]) dir_x=0; dir_y=1 ;;
    [Aa]) dir_x=-1; dir_y=0 ;;
    [Dd]) dir_x=1; dir_y=0 ;;
```

```
[Qq]) game_over=true ;;
esac

# UPDATE — move snake
new_x=$((snake_x + dir_x))
new_y=$((snake_y + dir_y))

# Check wall collision
if (( new_x < 0 || new_x >= WIDTH || new_y < 0 || new_y >= HEIGHT )); then
game_over=true
continue
fi

# Update body (add old head position)
body+=("$snake_x,$snake_y")

# Move head
snake_x=$new_x
snake_y=$new_y

# Check food collision
if [[ $snake_x -eq $food_x && $snake_y -eq $food_y ]]; then
((score++))
place_food
else
# Remove tail (keep body length proportional to score)
if (( ${#body[@]} > score )); then
body=("${body[@]:1}")
fi
fi
done
```

This Snake game combines everything: real-time loops, arrays for the snake body, collision detection, random food placement, colored output, raw terminal mode, and cleanup traps. It is a complete, playable arcade game in pure Bash.

Chapter 21

Terminal Game Projects

What You Will Learn

Six complete, playable terminal games that bring together everything you have learned. Each project

demonstrates different game mechanics and Bash techniques. Study, modify, and extend them.

Project 1: Number Guessing Game

```
#!/bin/bash
# guess.sh — number guessing game

target=$((RANDOM % 100 + 1))
attempts=0
max_attempts=7

echo "═══ Number Guess ═══"
echo "I'm thinking of a number between 1 and 100."
echo "You have $max_attempts attempts."
echo ""

while (( attempts < max_attempts )); do
((attempts++))
remaining=$((max_attempts - attempts + 1))
read -p "Attempt $attempts ($remaining left): " guess

# Validate input
if ! [[ "$guess" =~ ^[0-9]+$ ]]; then
echo "Please enter a number!"
((attempts--))
continue
fi

if (( guess < target )); then
echo "Too low!"
elif (( guess > target )); then
echo "Too high!"
else
echo "Correct! You guessed it in $attempts tries!"
echo "Score: $((100 - attempts * 10))"
exit 0
fi
done

echo ""
echo "Game Over! The number was $target."
```

Project 2: Quiz Game

#!/bin/bash

quiz.sh — trivia quiz game

score=0

```
total=0
# Questions array (question|answer|choices)
questions=(
    "What is the capital of France?|Paris|London|Berlin|Madrid"
    "Which planet is the largest?|Jupiter|Earth|Mars|Venus"
    "What is 15 * 12?|180|150|175|165"
    "Who wrote Romeo and Juliet?|Shakespeare|Dickens|Hemingway|Twain"
    "What is the chemical symbol for gold?|Au|Ag|Gd|Go"
    "What year did WW2 end?|1945|1939|1941|1950"
    "What is the speed of light (km/s)?|300000|150000|1000000|500000"
    "How many continents are there?|7|5|6|8"
)
echo "═══ Trivia Quiz ═══"
echo "${#questions[@]} questions. Good luck!"
echo ""
for q in "${questions[@]}"; do
    ((total++))
    IFS='|' read -r question correct opt1 opt2 opt3 <<< "$q"

    # Shuffle options
    options=("$correct" "$opt1" "$opt2" "$opt3")
    # Simple shuffle
    for i in ${!options[@]}; do
    j=$((RANDOM % ${#options[@]}))
    tmp="${options[$i]}"
    options[$i]="${options[$j]}"
    options[$j]="$tmp"
    done

    echo "Q$total: $question"
    for i in 0 1 2 3; do
    echo " $((i+1)). ${options[$i]}"
    done

    read -p "Your answer (1-4): " answer

    if [[ "$answer" =~ ^[1-4]$ ]] && [[ "${options[$((answer-1))]}" == "$correct" ]]; then
    echo " Correct!"
    ((score++))
    else
    echo " Wrong! Answer: $correct"
    fi
    echo ""
done
echo "═══ Results ═══"
echo "Score: $score / $total"
pct=$((score * 100 / total))
echo "Percentage: $pct%"
if (( pct >= 80 )); then
    echo "Excellent!"
```

elif ((pct >= 50)); then

```
    echo "Good job!"
```

else

```
    echo "Keep practicing!"
```

fi

Project 3: Hangman

#!/bin/bash

hangman.sh — hangman word game

Word list

words=(

```
    "ELEPHANT" "COMPUTER" "JAVASCRIPT" "BASKETBALL"
    "ADVENTURE" "KEYBOARD" "MOUNTAIN" "RAINBOW"
    "DRAGON" "CASTLE" "WIZARD" "POTION"
    "TREASURE" "DUNGEON" "SWORD" "SHIELD"
```

)

Pick random word

word="${words[$((RANDOM % ${#words[@]}))]}"

word_len=${#word}

guessed=""

wrong=0

max_wrong=6

Initialize display

display=""

for ((i=0; i<word_len; i++)); do

```
    display+="_"
```

done

echo "═══ Hangman ═══"

echo "Guess the word! ($word_len letters)"

draw_hangman() {

```
    local stage=$1
    case $stage in
    0)
    echo "  +---+"
    echo "  |   |"
    echo "      |"
    echo "      |"
    echo "      |"
    echo "      |"
    echo "========="
    ;;
    1)
    echo "  +---+"
    echo "  |   |"
    echo "  O   |"
    echo "      |"
    echo "      |"
    echo "      |"
    echo "========="
    ;;
    2)
    echo "  +---+"
    echo "  |   |"
    echo "  O   |"
    echo "  |   |"
    echo "      |"
    echo "      |"
    echo "========="
    ;;
    3)
    echo "  +---+"
    echo "  |   |"
    echo "  O   |"
    echo " /|   |"
    echo "      |"
    echo "      |"
    echo "========="
    ;;
    4)
    echo "  +---+"
    echo "  |   |"
    echo "  O   |"
    echo " /|\  |"
    echo "      |"
    echo "      |"
```

```
    echo "========="
    ;;
    5)
    echo "  +---+"
    echo "  |   |"
    echo "  O   |"
    echo " /|\  |"
    echo " /    |"
    echo "      |"
    echo "========="
    ;;
    6)
    echo "  +---+"
    echo "  |   |"
    echo "  O   |"
    echo " /|\  |"
    echo " / \  |"
    echo "      |"
    echo "========="
    ;;
    esac
}

while (( wrong < max_wrong )); do
```

```
echo ""
draw_hangman $wrong
echo ""
echo "Word: $(echo $display | sed 's/./& /g')"
echo "Guessed: $guessed"
echo "Wrong: $wrong/$max_wrong"
echo ""

read -n 1 -p "Guess a letter: " letter
letter=$(echo "$letter" | tr 'a-z' 'A-Z')
echo ""

# Check if already guessed
if [[ "$guessed" == *"$letter"* ]]; then
echo "Already guessed '$letter'!"
continue
fi

guessed+="$letter"

# Check if letter is in word
if [[ "$word" == *"$letter"* ]]; then
echo "Correct!"
# Update display
new_display=""
for ((i=0; i<word_len; i++)); do
if [[ "${word:$i:1}" == "$letter" ]]; then
new_display+="$letter"
else
new_display+="${display:$i:1}"
fi




done
display="$new_display"

# Check win
if [[ "$display" == "$word" ]]; then
echo ""
echo "═══ You Win! ═══"
echo "The word was: $word"
echo "Wrong guesses: $wrong"
exit 0
```

```
fi
else
echo "Wrong!"
((wrong++))
fi
done

echo ""
draw_hangman 6
echo ""
echo "═══ Game Over ═══"
echo "The word was: $word"
```

Project 4: Tic-Tac-Toe

```
#!/bin/bash
# tictactoe.sh — two-player tic-tac-toe

board=(" " " " " " " " " " " " " " " " " ")
current_player="X"
moves=0

draw_board() {
clear
echo "═══ Tic-Tac-Toe ═══"
echo ""
echo " ${board[0]} | ${board[1]} | ${board[2]} "
echo "───┼───┼───"
echo " ${board[3]} | ${board[4]} | ${board[5]} "
echo "───┼───┼───"
echo " ${board[6]} | ${board[7]} | ${board[8]} "
echo ""
echo "Player $current_player's turn"
echo "Positions 1-9 (top-left to bottom-right)"
}

check_win() {
local p=$1
local wins=(
"0 1 2" "3 4 5" "6 7 8"    # rows
"0 3 6" "1 4 7" "2 5 8"    # columns
"0 4 8" "2 4 6"             # diagonals
)
for combo in "${wins[@]}"; do
read -r a b c <<< "$combo"




if [[ "${board[$a]}" == "$p" && "${board[$b]}" == "$p" && "${board[$c]}" == "$p" ]]; then
return 0 # win
fi
done
return 1    # no win
```

}

while true; do

```
draw_board

read -p "Enter position (1-9): " pos

# Validate input
if ! [[ "$pos" =~ ^[1-9]$ ]]; then
echo "Invalid input! Use 1-9."
read -n 1 -s
continue
fi

((pos--))    # Convert to 0-indexed

if [[ "${board[$pos]}" == " " ]]; then
echo "Position taken!"
read -n 1 -s
continue
fi

board[$pos]=$current_player
((moves++))

# Check for win
if check_win "$current_player"; then
draw_board
echo "═══ Player $current_player Wins! ═══"
break
fi

# Check for draw
if (( moves == 9 )); then
draw_board
echo "═══ It's a Draw! ═══"
break
fi

# Switch player
if [[ "$current_player" == "X" ]]; then
current_player="O"
else
current_player="X"
fi
```

done

Project 5: Text Adventure

```
#!/bin/bash
# adventure.sh — text adventure game

# ── World Map (4 rooms) ──────────────────────────
# Room layout:
#   [1: Entrance] ←→ [2: Hallway] ←→ [3: Treasure Room]
#                         ↓
#                   [4: Dark Cave]

declare -A rooms
rooms[1,name]="Entrance"
rooms[1,desc]="You stand at the entrance of an ancient dungeon. A heavy stone door leads north."
rooms[1,north]=2

rooms[2,name]="Hallway"
rooms[2,desc]="A long, dark hallway. Exits: north, south, down."
rooms[2,north]=3
rooms[2,south]=1
rooms[2,down]=4

rooms[3,name]="Treasure Room"
rooms[3,desc]="Golden light fills the room. A chest sits in the center."
rooms[3,south]=2
rooms[3,item]="golden_key"

rooms[4,name]="Dark Cave"
rooms[4,desc]="A damp, dark cave. Something glints in the corner."
rooms[4,up]=2
rooms[4,item]="magic_sword"

# ── Player State ─────────────────────────────────
current_room=1
declare -A inventory
inventory["golden_key"]=0
inventory["magic_sword"]=0
has_treasure=false
moves=0

# ── Functions ────────────────────────────────────
show_room() {
echo ""
echo "═══ ${rooms[$current_room,name]} ═══"
echo "${rooms[$current_room,desc]}"

# Show item if present and not taken
```

```
if [[ -n "${rooms[$current_room,item]}" ]]; then
local item="${rooms[$current_room,item]}"
if (( inventory[$item] == 0 )); then
echo "You see a $item here."
fi
fi

# Show available exits
local exits=""
[[ -n "${rooms[$current_room,north]}" ]] && exits+="north "




[[ -n "${rooms[$current_room,south]}" ]] && exits+="south "
[[ -n "${rooms[$current_room,east]}" ]] && exits+="east "
[[ -n "${rooms[$current_room,west]}" ]] && exits+="west "
[[ -n "${rooms[$current_room,up]}" ]] && exits+="up "
[[ -n "${rooms[$current_room,down]}" ]] && exits+="down "
echo "Exits: $exits"
```

}

take_item() {

```
local item="${rooms[$current_room,item]}"
if [[ -n "$item" ]] && (( inventory[$item] == 0 )); then
inventory[$item]=1
echo "You take the $item."
# Remove item from room
rooms[$current_room,item]=""
else
echo "There's nothing to take here."
fi
```

}

show_inventory() {

```
echo "═══ Inventory ═══"
local found=false
for item in "${!inventory[@]}"; do
if (( inventory[$item] > 0 )); then
echo " - $item"
found=true
fi
done
[[ "$found" == "false" ]] && echo " (empty)"
```

}

── Main Game Loop ──────────────────────────────────────

echo "═══ The Dungeon of Destiny ═══"

echo "Type 'help' for commands."

echo ""

show_room

while true; do

```
echo ""
read -r -p "> " cmd arg

case "$cmd" in
n|north|go\ north)
if [[ -n "${rooms[$current_room,north]}" ]]; then
current_room=${rooms[$current_room,north]}
show_room
((moves++))
else
echo "You can't go north."
fi
;;
s|south|go\ south)
if [[ -n "${rooms[$current_room,south]}" ]]; then
current_room=${rooms[$current_room,south]}
show_room
((moves++))
else




echo "You can't go south."
fi
;;
d|down|go\ down)
if [[ -n "${rooms[$current_room,down]}" ]]; then
current_room=${rooms[$current_room,down]}
show_room
((moves++))
else
echo "You can't go down."
fi
;;
u|up|go\ up)
if [[ -n "${rooms[$current_room,up]}" ]]; then
current_room=${rooms[$current_room,up]}
show_room
((moves++))
else
echo "You can't go up."
fi
;;
take|get|pick)
```

```
take_item
;;
i|inv|inventory)
show_inventory
;;
open)
if [[ "$current_room" == "3" && "${inventory[golden_key]}" == "1" ]]; then
echo ""
echo "═══ VICTORY! ═══"
echo "You open the chest with the golden key!"
echo "Inside: mountains of gold and gems!"
echo "Moves: $moves"
exit 0
elif [[ "$current_room" == "3" ]]; then
echo "The chest is locked. You need a key."
else
echo "There's nothing to open here."
fi
;;
help|h|?)
echo "Commands: north/south/east/west/up/down, take, inventory, open, help, quit"
;;
quit|exit|q)
echo "Thanks for playing! Moves: $moves"
exit 0
;;
*)
echo "I don't understand '$cmd'. Type 'help'."
;;
esac
```

done

Project 6: Maze Runner

#!/bin/bash

maze.sh — navigate a maze to find the exit

Cleanup

cleanup() { stty sane; clear; }

trap cleanup EXIT

Maze layout (# = wall, . = path, P = player, E = exit)

maze=(

```
"###########"
"#P.......##"
"#.#####.#.#"
"#.#...#.#.#"
"#.#.#.#.#.#"
"#...#...#.#"
"#.##.##.#.#"
"#.....#...#"
"#####.#...#"
"#......#.E#"
"###########"
```

)

ROWS=${#maze[@]}

COLS=${#maze[0]}

Find player position

for ((r=0; r<ROWS; r++)); do

```
for ((c=0; c<COLS; c++)); do
if [[ "${maze[r]:$c:1}" == "P" ]]; then
player_r=$r
player_c=$c
fi
done
```

done

stty -echo -icanon

tput civis

while true; do

```
# RENDER
clear
echo "═══ Maze Runner ═══"
echo "Find the exit (E). WASD to move, Q to quit."
echo ""

for ((r=0; r<ROWS; r++)); do
for ((c=0; c<COLS; c++)); do
if [[ $r -eq $player_r && $c -eq $player_c ]]; then
printf "\033[32m@\033[0m"
else
char="${maze[r]:$c:1}"
case "$char" in
"#") printf "\033[90m▓\033[0m" ;;
"E") printf "\033[31mE\033[0m" ;;
*)   printf " " ;;




esac
fi
done
echo ""
done

# INPUT
read -n 1 -s key

# UPDATE
new_r=$player_r
new_c=$player_c

case "$key" in
[Ww]) ((new_r--)) ;;
[Ss]) ((new_r++)) ;;
[Aa]) ((new_c--)) ;;
[Dd]) ((new_c++)) ;;
[Qq]) echo "Quit!"; exit 0 ;;
esac

# Check collision
if (( new_r >= 0 && new_r < ROWS && new_c >= 0 && new_c < COLS )); then
char="${maze[new_r]:$new_c:1}"
if [[ "$char" == "#" ]]; then
```

```
player_r=$new_r
player_c=$new_c
if [[ "$char" == "E" ]]; then
clear
echo "═══ You Escaped! ═══"
echo "Congratulations! You found the exit!"
exit 0
fi
fi
fi
done
```

Extending the Games

Each project can be extended in many ways:

Game	Extensions
Number Guess	Difficulty levels, hints, high scores, multiplayer
Quiz	Categories, timer, lifelines, file-based questions
Hangman	Word categories, scoring, ASCII art, save/load
Tic-Tac-Toe	AI opponent, larger boards, win tracking
Adventure	More rooms, items, NPCs, combat, save/load
Maze	Randomly generated mazes, enemies, collectibles, timer

> Tip: Study and Modify

The best way to learn is to modify these games. Try adding features, changing mechanics, or combining elements from different projects. Every change teaches you something new about Bash.

> Project: Your Own Game

Now it's your turn. Design and build your own terminal game. Use the techniques from this book:

```
1. Design — What type of game? What are the rules?
2. State — What variables do you need?
3. Loop — Turn-based or real-time?
4. Input — How does the player interact?
5. Render — How do you display the game?
6. Win/Lose — What are the end conditions?
7. Polish — Add colors, sounds (with printf '\a' ), save/load
```

Start simple, then iterate. The journey from "Hello World" to a full game is shorter than you think.

Chapter 22

Debugging and Best Practices

What You Will Learn

How to find and fix bugs, write reliable scripts, and follow best practices that will make your Bash

games and scripts professional-grade.

Debugging Tools

set -x (Trace Execution)

```
# Enable tracing — prints each command before executing
set -x

# Trace specific section
set -x
# ... code to debug ...
set +x

# Trace with more detail (expand variables)
set -v    # verbose: print input lines as read
set -x    # xtrace: print commands after expansion

# Run a script with tracing without modifying it
bash -x game.sh

# Trace to a file
exec 2>debug.log
set -x
# ... commands ...
set +x
exec 2>&1
```

ShellCheck

ShellCheck is the #1 tool for catching Bash bugs. Install it and use it on every script:

```
# Install ShellCheck
# Ubuntu/Debian: sudo apt install shellcheck
# macOS: brew install shellcheck
# Online: https://www.shellcheck.net

# Check a script
shellcheck game.sh




# Check with wiki links
shellcheck game.sh

# Check in your editor (VS Code, Vim, etc.)
# Install the ShellCheck extension
```

Common ShellCheck warnings:

```
SC2086: Double quote variables to prevent globbing
echo $var          → echo "$var"

SC2046: Quote command substitution
cd $(dirname $0)   → cd "$(dirname "$0")"

SC2034: Variable appears unused
unused_var="hi"    → remove or use it

SC2155: Declare and assign separately
local var=$(cmd)   → local var; var=$(cmd)

SC2004: $/${} on arithmetic variables
echo $(( $a + 1 )) → echo $(( a + 1 ))

SC2181: Check exit code directly
cmd; if [ $? -eq 0 ] → if cmd; then

SC2230: Use command -v instead of which
which bash           → command -v bash
```

Common Bash Bugs

1. Unquoted Variables

```
# BUG: Word splitting on spaces, glob expansion on *
for file in $files; do ...    # Breaks if filenames have spaces

# FIX: Always quote
for file in "$files"; do ... # Still wrong (single string)
for file in "${files[@]}"; do ... # Correct for arrays
```

2. Forgetting then/fi/do/done

```
# BUG: Missing keywords
if [[ $x == "y" ]]     # Missing 'then'
echo "yes"
fi

# FIX:




if [[ $x == "y" ]]; then
echo "yes"
fi

# Same for loops
for i in 1 2 3; do     # Missing 'do'
echo $i
done
```

3. Using = Instead of -eq for Numbers

```
# BUG: String comparison, not numeric
if [[ "$count" = "5" ]]; then ...   # Works but wrong semantics

# FIX: Use arithmetic comparison
if (( count == 5 )); then ...
if [[ "$count" -eq 5 ]]; then ...
```

4. CRLF Line Endings

```
# BUG: Script created on Windows has \r\n line endings
# Error: $'\r': command not found

# FIX: Convert to Unix line endings
dos2unix script.sh
# or
sed -i 's/\r$//' script.sh
# or in vim: :set fileformat=unix
```

5. Subshell Variable Loss

```
# BUG: Variables set in pipe subshell are lost
count=0
echo -e "a\nb\nc" | while read line; do
count=$((count + 1))
done
echo $count     # 0 — lost!

# FIX: Use process substitution
count=0
while read line; do
count=$((count + 1))
done < <(echo -e "a\nb\nc")
echo $count    # 3 — correct!
```

6. Forgetting to Export Functions

```
# BUG: Function not available in subshell
my_func() { echo "hi"; }
bash -c "my_func"   # Error: my_func: command not found

# FIX: Export the function
my_func() { echo "hi"; }
export -f my_func
bash -c "my_func"   # Works!
```

Error Handling

```
#!/bin/bash
set -euo pipefail

# Error handler function
error_handler() {
local line=$1
echo "Error on line $line"
# Cleanup code here
exit 1
}
trap 'error_handler $LINENO' ERR

# Check prerequisites
check_command() {
if ! command -v "$1" &>/dev/null; then
echo "Error: '$1' is not installed." >&2
exit 1
fi
}

check_command "grep"
check_command "awk"
check_command "sed"

# Safe file operations
safe_read() {
local file="$1"
if [[ ! -f "$file" ]]; then
echo "Error: File not found: $file" >&2
return 1
fi
if [[ ! -r "$file" ]]; then
echo "Error: Cannot read file: $file" >&2
return 1
fi
cat "$file"
}

# Safe division (avoid division by zero)
safe_divide() {
local a=$1 b=$2
```

```
if (( b == 0 )); then
echo "Error: Division by zero" >&2
return 1
fi
echo $(( a / b ))
}
```

Traps for Cleanup

```
# Cleanup on exit (always runs, even on Ctrl+C)
cleanup() {
# Restore terminal
stty sane
tput cnorm

# Remove temp files
rm -f /tmp/game_*.tmp

# Save game state
if [[ "$game_started" == "true" ]]; then
save_game
fi

clear
echo "Game exited."
}

trap cleanup EXIT
trap cleanup INT         # Ctrl+C
trap cleanup TERM        # kill command
```

Best Practices Checklist

Practice	Why
Start with #!/bin/bash	Ensures correct interpreter
Use set -euo pipefail	Catch errors early
Quote all variables	Prevent word splitting
Use local in functions	Avoid variable leaks
Use [[ ]] over [ ]	More features, safer
Use $() over backticks	Readable, nestable
Use ${var} over $var	Unambiguous
Use (( )) for arithmetic	Cleaner than $(( ))
Add comments	Future you will thank you

Use functions	Reusable, organized code
Use trap for cleanup	Always restore terminal
Run ShellCheck	Catch bugs before they bite
Validate user input	Prevent crashes and exploits
Use meaningful names	player_hp, not x
Handle errors gracefully	Better UX, easier debugging

Exercise: Debugging

```
1. Run bash -x on one of the game scripts and study the trace output.
2. Install ShellCheck and run it on your scripts.
3. Intentionally introduce a bug (remove quotes) and see what ShellCheck reports.
4. Add a trap to a game script that saves on exit.
5. Add input validation to prevent non-numeric input in the guessing game.
```

Common Mistake: Not Testing Edge Cases

```
Always test edge cases in your games:

• What if the player enters nothing (empty input)?
• What if they type a letter when you expect a number?
• What if the save file is corrupted or missing?
• What if the terminal is too small?
• What if the player presses Ctrl+C mid-game?

Handle these gracefully with validation, error messages, and cleanup traps.
```

Project: Game Shell Template

```
Create a robust game template you can reuse for all future projects:


#!/bin/bash
#
# game_template.sh — reusable game template
# Usage: ./game_template.sh [player_name]
#

# ── Safety Settings ──────────────────────────────────
```

set -euo pipefail

── Cleanup ──

cleanup() {

```
stty sane 2>/dev/null
tput cnorm 2>/dev/null
rm -f "$TMPFILE" 2>/dev/null
```

}

trap cleanup EXIT INT TERM

── Constants ──

GAME_NAME="My Game"

VERSION="1.0"

TMPFILE=$(mktemp)

── Colors ──

RED='\033[0;31m'

GREEN='\033[0;32m'

YELLOW='\033[1;33m'

CYAN='\033[0;36m'

BOLD='\033[1m'

NC='\033[0m'

── Game State ──

player_name="${1:-Adventurer}"

player_hp=100

player_gold=0

score=0

game_over=false

── Utility Functions ──────────────────────────────────────

die() { echo -e "${RED}Error: 1{NC}" >&2; exit 1; }

msg() { echo -e "${CYAN}$1${NC}"; }

ok()	{ echo -e "${GREEN}$1${NC}"; }

warn() { echo -e "${YELLOW}$1${NC}"; }

pause() { read -n 1 -s -r -p "Press any key to continue..." _; echo; }

clear_screen() { printf '\033[H\033[2J'; }

── Input Validation ───────────────────────────────────────

read_number() {

```
local prompt="$1"
local min="$2" max="$3"
while true; do
read -rp "$prompt" input
if [[ "$input" =~ ^[0-9]+$ ]] && \
(( input >= min && input <= max )); then
echo "$input"
return 0
fi
warn "Please enter a number between $min and $max."
done
```

}

── Main Game ──

main() {

```
clear_screen




echo -e "${BOLD}${CYAN}═══ $GAME_NAME v$VERSION ═══${NC}"
echo ""
msg "Welcome, $player_name!"
pause

# Game loop
while [[ "$game_over" == "false" ]]; do
clear_screen
echo "HP: $player_hp | Gold: $player_gold | Score: $score"
echo ""
echo "1. Adventure"
echo "2. Shop"
echo "3. Status"
echo "4. Quit"

choice=$(read_number "Choose (1-4): " 1 4)

case "$choice" in
1) msg "Going on an adventure!" ;;
2) msg "Welcome to the shop!" ;;
3) msg "HP: $player_hp | Gold: $player_gold | Score: $score" ;;
4) game_over=true ;;
esac
pause
done

ok "Thanks for playing $GAME_NAME!"
echo "Final score: $score"
}

main "$@"
```

This template includes safety settings, cleanup traps, colors, input validation, utility functions, and

a game loop structure. Start every new game by copying this template and filling in the details.

Chapter 23

References and Resources

What You Will Learn

Where to go next. This chapter provides curated references, documentation links, and learning resources to continue your Bash journey.

Official Documentation

1. GNU Bash Reference Manual — the complete, authoritative guide to every Bash feature.

```
https://www.gnu.org/software/bash/manual/bash.html
```

2. GNU grep Manual — documentation for grep and regular expressions.

```
https://www.gnu.org/software/grep/manual/grep.html
```

3. GNU sed Manual — complete sed reference with examples.

```
https://www.gnu.org/software/sed/manual/sed.html
```

4. GNU awk (gawk) Manual — comprehensive awk programming guide.

```
https://www.gnu.org/software/gawk/manual/gawk.html
```

5. The Linux man-pages project — system manual pages for all core utilities.

```
https://www.kernel.org/doc/man-pages/
```

6. ShellCheck — static analysis tool and wiki for Bash scripting.

```
https://www.shellcheck.net/
```

7. ShellCheck Wiki — detailed explanations of common shell scripting issues.

```
https://github.com/koalaman/shellcheck/wiki
```

Learning Resources

1. Bash Hackers Wiki — community-driven Bash knowledge base.

```
https://wiki.bash-hackers.org/
```

2. Greg's Wiki (Wooledge.org) — practical Bash advice from a Bash expert.

```
https://mywiki.wooledge.org/
```

3. The Advanced Bash-Scripting Guide — classic in-depth Bash tutorial.

```
https://tldp.org/LDP/abs/html/
```

4. Learn Bash the Hard Way — interactive, exercise-based learning.

```
https://learnbashthehardway.org/
```

5. explainshell.com — paste a command, get an explanation of every flag.

```
https://explainshell.com/
```

6. shellcheck on GitHub — source code and issue tracker.

```
https://github.com/koalaman/shellcheck
```

Quick Reference Cards

Variable Syntax

Syntax	Description
${var}	Value of var
${var:-default}	Use default if var is unset/empty
${var:=default}	Assign default if var is unset/empty
${#var}	Length of var
${var:offset:length}	Substring
${var/old/new}	Replace first occurrence
${var-/old/new}	Replace all occurrences
${var#pattern}	Remove shortest prefix
${var##pattern}	Remove longest prefix
${var%pattern}	Remove shortest suffix
${var%%pattern}	Remove longest suffix
${var^^}	Uppercase all
${var,,}	Lowercase all

Test Operators

Operator	True if...
-e FILE	File exists
-f FILE	Regular file
-d FILE	Directory
-r FILE	Readable

-w FILE	Writable
-x FILE	Executable
-s FILE	Non-empty file

-z STRING	Empty string
-n STRING	Non-empty string
N1 -eq N2	Numbers equal
N1 -ne N2	Numbers not equal
N1 -lt N2	N1 less than N2
N1 -gt N2	N1 greater than N2
N1 -le N2	N1 less or equal
N1 -ge N2	N1 greater or equal

Special Variables

Variable	Meaning
$0	Script name
$1 - $9	Positional parameters
$#	Number of arguments
$@	All arguments (quoted separately)
$*	All arguments (single string)
$?	Exit status of last command
$$	Current PID
$!	PID of last background job
$RANDOM	Random number 0-32767
$LINENO	Current line number
$BASH_REMATCH	Regex match results

ANSI Color Codes

Code	Color
\033[0;30m	Black
\033[0;31m	Red

\033[0;32m	Green
\033[0;33m	Yellow
\033[0;34m	Blue
\033[0;35m	Magenta
\033[0;36m	Cyan
\033[0;37m	White

\033[1m	Bold
\033[4m	Underline
\033[0m	Reset

Books and Further Reading

1. "The Linux Command Line" by William Shotts — free online book covering Linux commands and shell

```
scripting.
https://linuxcommand.org/tlcl.php
```

2. "Bash Guide for Beginners" by Machtelt Garrels — Linux Documentation Project guide.

```
https://tldp.org/LDP/Bash-Beginners-Guide/html/
```

3. "Pure Bash Bible" — community collection of pure Bash alternatives to external processes.

```
https://github.com/dylanaraps/pure-bash-bible
```

Community

1. r/bash on Reddit — Q&A, tips, and project sharing.

```
https://www.reddit.com/r/bash/
```

2. Stack Overflow — Bash tag with hundreds of thousands of questions.

```
https://stackoverflow.com/questions/tagged/bash
```

3. Bash subreddit for shell script games and challenges.

```
https://www.reddit.com/r/commandline/
```

What's Next?

You now have the skills to build terminal games in Bash. Here are paths to continue growing:

```
• Master the tools — Dive deeper into sed and awk; they are Turing-complete languages.
```

• Learn other shells — Try zsh or fish for different features and ergonomics.

• Explore other languages — Python, Rust, or Go for more complex terminal applications.

• Build bigger games — Combine multiple scripts, use ncurses for advanced TUIs, add networking for multiplayer.

• Contribute — Share your games on GitHub, help others learn Bash.

• Automate everything — Apply your skills to system administration, CI/CD, and automation.

> Tip: The Terminal Is Your Canvas

Bash is more than a shell — it is a creative platform. With the tools and techniques in this book, you can build games, tools, and experiences that run on virtually any computer in the world. The

terminal is your canvas, and Bash is your brush. Now go build something amazing.

```
— End of "Mastering Bash: From Beginner to Terminal Game Developer" —
```